AFIT/GE/ENG/95M-01

ANALYSIS AND EXTENSION OF MODEL-BASED SOFTWARE EXECUTIVES

THESIS
Keith E. Lewis
Captain, USAF

AFIT/GE/ENG/95M-01

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 1995	Master's Thesis

4. TITLE AND SUBTITLE
ANALYSIS AND EXTENSION OF MODEL-BASED SOFTWARE EXECUTIVES

5. FUNDING NUMBERS

6. AUTHOR(S)
Keith E. Lewis

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GE/ENG/95M-01

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Capt Rick Painter
2241 Avionics Circle, Suite 16
WL/AAWA-1 BLD 620
Wright-Patterson AFB, OH 45433-7765

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
This research developed a comprehensive description of the simulation environment of Architect, a domain-oriented application composition system being developed at the Air Force Institute of Technology to explore new software engineering technologies. The description combines information from several previous research efforts and Architect's source code into a single, comprehensive document. A critical evaluation of the simulation environment was also performed, identifying improvements and modifications that enhance Architect's application execution capabilities by reducing complexity and execution time. The analysis was then taken one step further and presented extensions to the current simulation environment. The extensions included investigating the feasibility of mixed-mode execution and integrating the composition of application executives with the visual system interface.

14. SUBJECT TERMS
software engineering, executive, application executive, simulation executive, application composition systems, domain modeling

15. NUMBER OF PAGES
100

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

AFIT/GE/ENG/95M-01

ANALYSIS AND EXTENSION OF MODEL-BASED SOFTWARE EXECUTIVES

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Keith E. Lewis, B.S.E.E.

Captain, USAF

December 13, 1994

Acknowledgements

I would first like to thank my thesis committee members, Major Paul Bailor, Dr. Thomas Hartrum, and Dr. Eugene Santos. I would like to give a special thanks to Maj Bailor for his help, patience, and advice. His guidance enabled me to persevere when, at times, it didn't seem possible. I would also like to thank Dan Zambon and Dave Doak for their unceasing efforts in keeping the computer systems up and running.

I would like to thank my fellow KBSE researchers for all the support they gave me during this research. They helped to make my AFIT experience both tolerable and enjoyable. A special thanks goes to Rich Guinto for his support and friendship. He helped me more than he'll ever know just by listening and offering suggestions.

Finally, I would like to thank my family for their faith in me and their eternal support. I would like to thank my parents for instilling in me the importance of education and teaching me to always do my best. I want to thank my children, Danielle and Kevin, for understanding why I couldn't spend the time with them that they deserved over the past 18 months. Most of all, I want to thank my wife and best friend, Darinda, for all the personal sacrifices she has made for me. Her undying love, understanding, encouragement, and faith throughout this AFIT program, and all my endeavors, keeps me going through all of my difficulties and successes. It is to her I dedicate this thesis effort.

Keith E. Lewis

Table of Contents

List of Figures

Abstract

This research developed a comprehensive description of the simulation environment of Architect, a domain-oriented application composition system being developed at the Air Force Institute of Technology to explore new software engineering technologies. The description combines information from several previous research efforts and Architect's source code into a single, comprehensive document. A critical evaluation of the simulation environment was also performed, identifying improvements and modifications that enhance Architect's application execution capabilities by reducing complexity and execution time. The analysis was then taken one step further and presented extensions to the current simulation environment. The extensions included investigating the feasibility of mixed-mode execution and integrating the composition of application executives with the visual system interface.

ANALYSIS AND EXTENSION OF MODEL-BASED SOFTWARE EXECUTIVES

I. Introduction

1.1 Background

The Software Engineering Institute (SEI) at Carnegie-Mellon University studied the idea of model-based software development in their Software Architectures Engineering (SAE) project. They developed the Object-Connection-Update (OCU) model which describes a set of reusable software building blocks (16). These building blocks can be combined to form system models, thus reusing design knowledge from previous successful implementations of the building blocks. In addition, the behavior of the system can be derived from the known behavior of the lower-level building blocks. A detailed description of the OCU model is presented in Chapter II.

Model-based software development encourages software engineers to view software development in the same way that traditional engineering disciplines view product development. Traditional engineering disciplines have long recognized and successfully employed reuse of design products. They benefit from large bodies of scientific knowledge, which over the years, have been codified into models (8). These models act as reusable templates from which to construct practical, working solutions to problems in a specific engineering area. For example, "automotive engineers have models of cars, civil engineers have models of bridges, mechanical engineers have models of rolling mills, electrical engineers have models

of motors ..." (8:140). The engineer is trained to understand these models, to recognize which model will solve a particular problem, and to adapt the model, if necessary, to fit a specific application.

Software engineering is a relatively new discipline which is just now beginning to codify its body of knowledge. Unlike other engineering disciplines, it does not currently rely on models as a means of designing working solutions. Instead, each new problem is treated in isolation as a unique situation requiring a unique solution. D'Ippolito maintains that "models can do for the software industry what they have done before and continue to do for the main-line engineering professions" (7:258). That is, they can provide "reuse at the design level, reduced system complexity, a means to measure project risk, reduced coding costs, reduced testing costs, reduced documentation costs, and increased maintainability and enhanceability" (7:258).

Another benefit of model-based software components is the ability to perform simulations on the models. The Knowledge-Based Software Engineering (KBSE) group at the Air Force Institute of Technology (AFIT) is researching this technology and is in its third year of funding by the Joint Modeling and Simulation System (J-MASS) program. The J-MASS program is intended to provide a flexible, standardized, and validated modeling tool to support a wide range of simulation requirements across the life cycle of a weapon system (3). J-MASS project members intend to apply a version of the OCU design methodology to create and assemble simulation applications. The Air Force will eventually use these models for simulations in many critical application domains (33).

Within an application domain, there are instantiated models of low-level reusable components that can be put together to form a high-level system. Since the behavior of the low-level components is known, the overall behavior of the system is known and can be tested as it responds to various simulation scenarios. For example, in the cruise missile domain, a cruise missile may consist of a propulsion subsystem, an airframe subsystem, an avionics subsystem, and a warhead subsystem. The knowledge base for this domain would have several models (components) for each of the different subsystems. With such models, an engineer assigned the task of designing a new cruise missile to meet a given set of requirements could relatively easily compose cruise missiles with different subsystems, simulate them, and evaluate their performance against the given requirements.

1.1.1 Architect. To investigate the feasibility of model-based software development, the KBSE group at AFIT is developing a prototype, domain-oriented, application composition system called Architect. The purpose of Architect is to allow an application specialist to input a system specification in a domain-specific language, verify the behavior of the specification, and transform the specification into a target language. An application specialist is a "sophisticated user" that is familiar with the overall domain and understands what the application must do to meet requirements (23). After verifying (through simulation) that the new composition behavior meets the specification, the ultimate goal is to use Architect to synthesize the appropriate implementation of the specification (i.e. transform the specification into a target language).

The simulated executions used for verification are controlled by an application executive. The application executive is a supervisory program similar to an operating system

kernel. As such, it oversees the operation of all software routines, manages all the resources of the application, and provides the interface between the software and the hardware (32). The application executive is a vital subsystem within Architect and was the primary focus of this research.

1.1.2 Big Picture. Figure 1.1 shows the high-level design of the Architect system being constructed at AFIT. The application executive controls the execution of a composed application and is illustrated in the bottom left hand corner of the diagram.

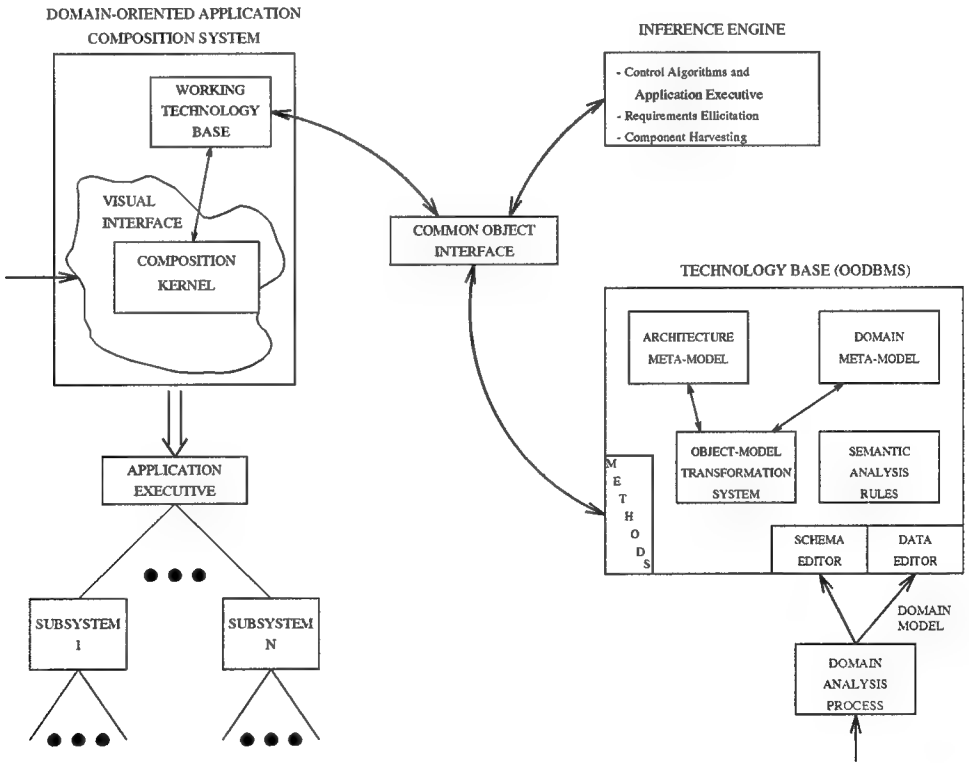

Figure 1.1 Domain-Oriented Application Composition Environment

1.2 Problem

Previous research (32) developed an application executive that supported three modes of execution: non-event-driven sequential, event-driven sequential, and time-driven sequential. Prior to composing an application, the application composer had to select the execution mode. Once the mode of execution was selected, all subsystems in the application were executed in this mode; there was no capability for independent subsystems within an application to execute in different modes. Furthermore, the previous implementation of Architect did not support composable application executives. The event-driven and time-driven executives were instantiated textually and saved to a file. Each time a new application was created, the corresponding executive was parsed into the application. There was no way to use the Architect Visual System Interface (AVSI) to compose the application executive.

In addition, the information explaining the design and implementation of the application executive was contained in several different research efforts (9, 29, 32). This information was not available in a single document, making it very difficult to comprehend. Also, there were some inconsistencies between the various sources which did little to aid understanding.

1.2.1 Problem Statement.

Consolidate the information detailing the implementation of the application executive. Expand the existing implementation of Architect to allow independent subsystems to execute in different modes and provide composable application executives.

1.3 Scope

This research focuses on analyzing the implementation of Architect's application executive and identifying improvements and/or modifications that would enhance Architect's simulation capabilities. In addition, this research concentrates on the event-driven sequential and time-driven sequential modes of execution. It does provide a brief discussion of the non-event-driven sequential mode but does not attempt to develop the concurrent execution modes conceptualized by Welgan (32).

1.4 Research Objectives

The following research objectives were established to satisfy the problem statement in Section 1.2.1.

- To develop an indepth understanding of the current implementation of the application executive.
- To consolidate the information detailing the implementation of the application executive into a single, comprehensive document.
- To perform a critical evaluation of the application executive, analyzing the overall operational concept and the domain model for the event-driven and time-driven sequential operating modes.
- To analyze Architect to determine changes required for independent subsystems to execute in different modes.
- To complete the integration of the application executive with the composition kernel and integrate the application executive with AVSI.

1.5 Sequence of Presentation

The remainder of this thesis is organized as follows: Chapter II presents a more detailed background of the Architect system and the OCU model. It is important to provide this information in order for the reader to understand material developed in the remainder of this research. Chapter III contains a comprehensive description of the current implementation of the application executive. Chapter IV provides an analysis of the existing application executive. It looks at the operational concept and the domain model of the executive and identifies improvements/enhancements that can be applied to the current implementation. Chapter V identifies extensions that can be made to the current implementation of the application executive. In particular, it develops the changes required to support independent subsystems executing in different operating modes and integrating visually composable application executives. Finally, Chapter VI contains the conclusions of this research and recommendations for further research.

II. Literature Review

2.1 Introduction

Chapter I introduced the concept of model-based software development as a methodology that could provide software engineers with the ability to reuse design knowledge. Reuse at the design level reduces coding, testing, and documentation costs and increases maintainability and enhanceability. To investigate the feasibility of this approach, the Architect system, based on the Object-Connection-Update (OCU) model, was constructed. This chapter discusses several areas that relate to model-based software development and specifically to the Architect system. First, the OCU model is discussed, introducing the elements and functions of the model. Second, the Architect system itself is discussed, looking at domain-oriented application composition, its development environment, and its history. Third, an overview of the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) is presented. VHDL is a standardized and well documented language, and it provides a simulation environment similar to Architect's. Finally, a brief introduction to simulation is presented, highlighting system simulation and differentiating event-driven from time-driven simulation.

2.2 Object Connection Update (OCU) Model

As stated in Section 1.1, the OCU model was developed by the SEI. However, there exists no detailed description of OCU; the best description available is contained in an SEI draft report by Lee (16). The following sections summarize the information found in this report.

2.2.1 OCU Description. The OCU model is the software architecture of Architect. Lee states that a software architecture is "a selection, from a technology base, of models and composition rules that defines the structure, performance, and use of a system relative to a set of engineering goals" (16:8). In other words, an architecture is a way of composing models, using engineering practices to achieve an overall design.

The OCU model is especially well-suited for developing software systems that can be described as of a set of subsystems. The *subsystem* is the main element in the OCU model. A software system, or application, is described as a set of subsystems controlled by an executive function called an application executive. Figure 2.1 shows how a subsystem is represented in the OCU model.

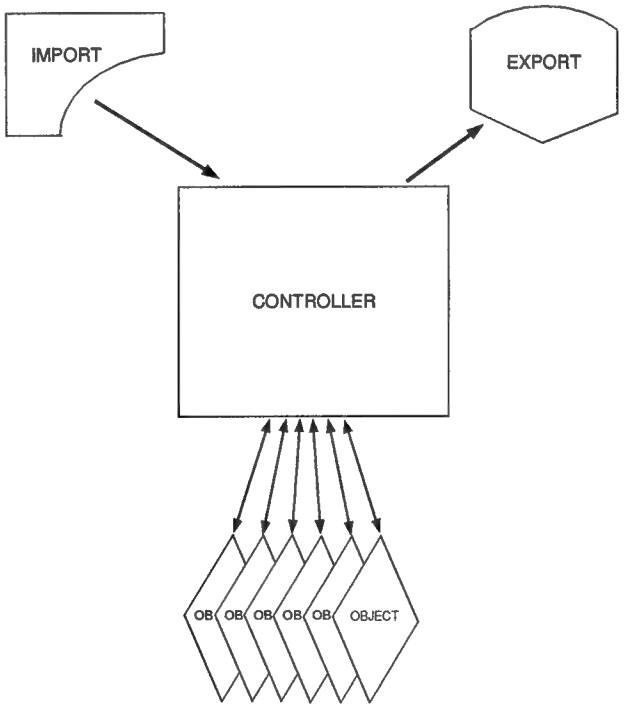

Figure 2.1 An OCU Subsystem

An OCU subsystem consists of one or more *objects*, an *import area*, an *export area*, and a *controller*. The objects model behavior of real-world components and maintain state. The objects are either primitive objects or other lower-level subsystems, thus allowing for nested subsystems. Each primitive object has an input area, an output area, and an associated set of algorithms for inputing new data, transforming the current state, and producing output data. The import area is the central collection point for the inputs needed by the objects in the subsystem. It collects data from other subsystems in the application and makes them available to its objects. The export area is the central distribution point for the outputs of a subsystem needed by other subsystems in the application. The controller manages the objects in the subsystem. It establishes connections between the objects and handles the flow of information to and from them, based on the function or purpose of the system.

In order to carry out its mission, each subsystem has a common set of functions. These functions are *Update, Stabilize, Initialize, Configure,* and *Destroy* (16). Update causes the controller to update the state of the subsystem based on the current state of the objects it controls and the input data in the import area; in addition, state data needed externally is written to the export area. Stabilize gets rid of transients; it causes the subsystem to reach a steady state consistent with its import data and current state. Initialize creates new objects and sets their initial states. Configure maps object inputs to the import area and object outputs to the export area (i.e., it defines how the subsystem is connected internally and externally). Destroy deallocates objects that were created during Initialize.

Similarly, there is a common set of functions for each object. These functions consist of *Update, Create, SetFunction, SetState,* and *Destroy* (16). Update causes the object to calculate a new state based on its current state and its inputs, Create instantiates a new instance of an object, SetFunction redefines or alters the algorithm used by the Update function, SetState changes an object's state directly, and Destroy deallocates objects.

2.2.2 Application Executive. An important aspect of the OCU model is anonymity. Subsystems know only about their objects and have no knowledge of other subsystems. Objects have no knowledge of other objects or of subsystems. Because of this, there needs to be some way to control the overall execution of the application. This is the role of the application executive.

In (16), Lee mentions the application executive only briefly. Even though it is not covered in detail, it is possible to infer some information about the application executive from the rest of the paper's description of the OCU model. The executive is modeled as an OCU subsystem and manages all the resources of an application for the purpose of controlling execution. It is a high-level supervisory subsystem that coordinates the behavior and information flow between the other subsystems in the application. Each subsystem in the OCU architecture is passive and waits for the application executive to tell it to perform one of its operations (16:18). For example, when the executive triggers (e.g., calls) the Update function of the controller of a subsystem, the subsystem (via calls to Update functions of the objects it controls) reads the data in its import area, reacts based on the data and its current state, and places any data that may be needed by other subsystems in its export area. The application executive must have knowledge of each

subsystem in the application, as it is also responsible for the mapping between import areas and export areas of separate subsystems and between the import/export areas and I/O devices.

2.3 Architect

2.3.1 Domain-Oriented Application Composition System. As stated in Section 1.1, Architect is a prototype domain-oriented application composition system. A domain-oriented application composition system has several distinguishing characteristics. It contains a knowledge base organized into domains and includes a process to compose applications. The knowledge base stores reusable components within an application domain. The components encapsulate domain knowledge, and applications are composed using these components. In Architect, the domain knowledge for each application domain is captured in its *technology base.* The knowledge base for the entire system is the aggregation of the individual, domain-specific, technology bases of all supported domains.

Through the composition process, the application composer can create, modify, save, and maintain applications. Once the application is composed, the environment provides the user with the ability to simulate and verify the behavior of the application and synthesize executable code. Architect doesn't currently support code synthesis; this has been recommended as an area of future research for the project. Figure 2.2 shows a big picture of a generic domain-oriented application composition system. The rounded boxes represent processes and the rectangular boxes represent physical structures. For a more detailed discussion of this type of system, see Warner (30).

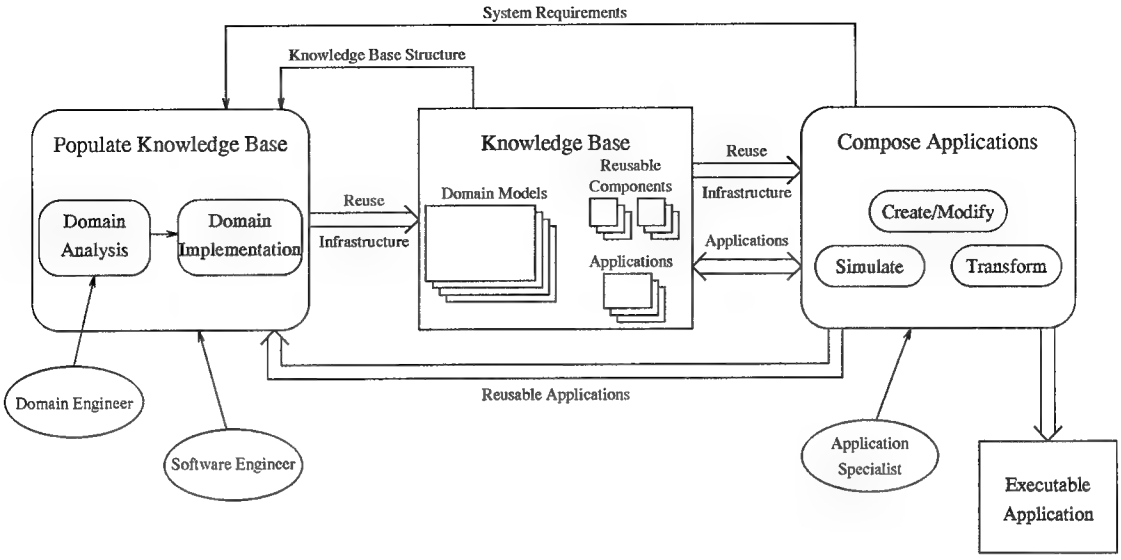

Figure 2.2 Generic Domain-Oriented Application Composition System(30:3-3)

2.3.2 Domain Analysis and Modeling. Section 2.3.1 introduced Architect's knowledge base as a collection of domain-specific technology bases. It is important to have a method, or process, to accurately and thoroughly capture the desired domain knowledge into the associated technology base. Domain analysis is the process that performs this function. Prieto-Díaz defines domain analysis as "...a process by which information used in developing software systems is identified, captured, and organized with the purpose of making it reusable when creating new systems" (21). Since the Architect system is the result of numerous research efforts, several different domain analysis techniques have been used to capture the domain knowledge encapsulated in its knowledge base (2, 14, 18, 19, 21, 22, 28).

2.3.3 Development Environment. Architect is based on the OCU architecture and is built within the Software Refinery environment. Software Refinery consists of the Refine^{TM} (26) wide spectrum language, the Dialect (25) language processing tool, and

the Intervista (24) graphical tool. As a wide-spectrum language, Refine^{TM} is capable of expressing an application at the specification level or the code level. At the specification level, it is necessary to only specify *what* transformations are needed to display the desired behavior. There are high-level abstractions such as set formers, rules, and transforms. Using these, the user can specify a set of post-conditions that must be made true whenever certain pre-conditions are true, without being concerned with how the post-conditions are met. At the code level, traditional programming constructs are available to allow the user to specify *how* to implement the desired behavior.

2.3.4 Implementation. The original implementation of Architect was developed by Anderson (1) and Randour (23) in 1992 and had only a textual interface. This initial version had a single application domain, digital logic circuits, and a rudimentary simulation capability, limited to simulating applications that exhibit non-event-driven-sequential behavior. Weide (31) extended Architect to include a graphical interface called the Architect Visual System Interface (AVSI). Subsequent research has been done by several KBSE members. Cossentine (6) developed an enhanced AVSI. Warner (30) created a method for populating the knowledge base with new domains and created a technology base for the digital signal processing domain. Welgan (32) extended the execution/simulation capability by incorporating an application executive that supported event-driven and time-driven applications. Gool (9) identified and implemented architectural changes necessary to support event-driven and time-driven applications. Waggoner (29) created a new technology base for the event-driven logic circuits domain and developed a cruise missile technology base to support the time-driven mode of execution. Finally, Cecil and Fullencamp (5)

implemented an object-oriented database to store the domain models in the knowledge base. In addition, there are two research efforts being conducted concurrently with this research: Guinto (11) is developing an enhanced AVSI, and Harris (12) is extending the database support.

Figure 2.3 shows a general overview of Architect. From this figure, it is easy to see the similarity between Architect and the generic domain-oriented application composition system of Figure 2.2. The square boxes represent the actual components of Architect. The

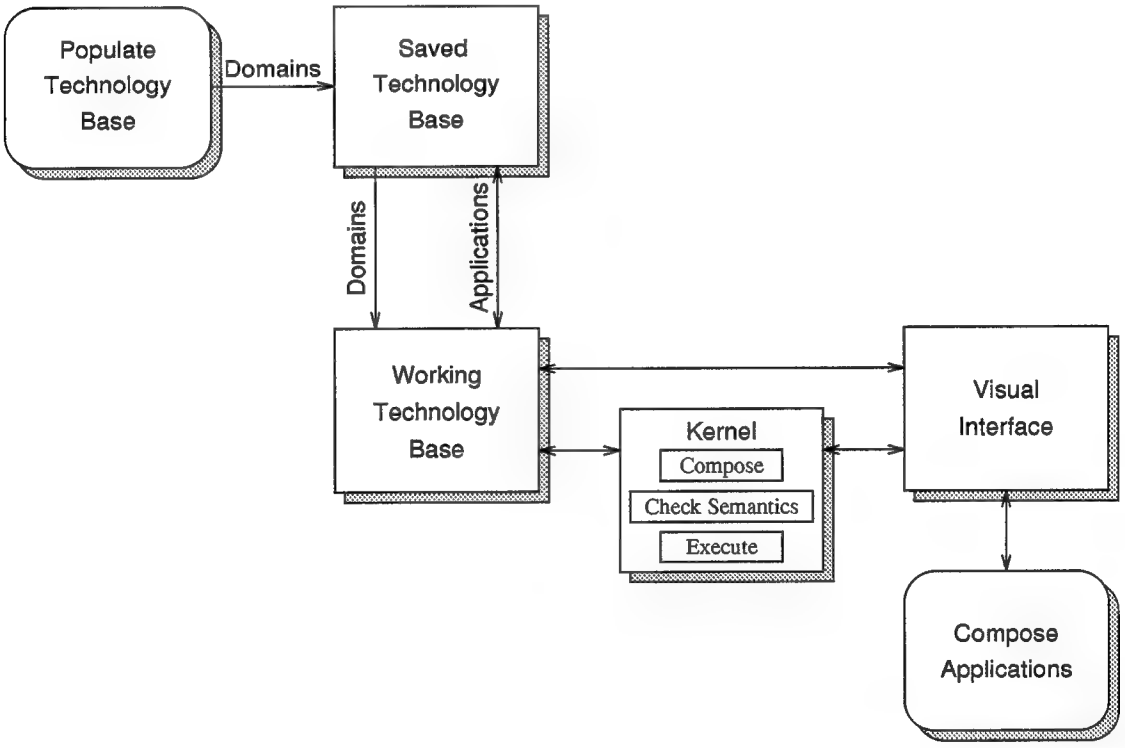

Figure 2.3 General System Overview of Architect(30:4-2)

rounded boxes represent tasks accomplished by the domain engineer, the software engineer, and/or application specialist.

The domain engineer analyzes the domain of interest to determine the domain knowledge that needs to be captured in the technology base. He works together with the software engineer to encapsulate this knowledge into models and populate the technology base. At the lowest level, these models are represented as primitive objects. At a higher level of abstraction, the models can be represented as subsystems composed of primitive objects and/or other subsystems. The application specialist uses these models along with the AVSI to view domain documentation and to compose, load, check the semantics of, execute, and save applications. Applications are composed of subsystems, and subsystems, once again, are composed of primitive objects and/or other, lower-level subsystems.

2.3.5 Application Execution. Application execution is controlled by the application executive. Architect currently supports three modes of execution: non-event-driven sequential, event-driven sequential, and time-driven sequential. In the non-event-driven sequential mode, each subsystem controller has an update algorithm which is a static list of function calls for its subordinate components. During execution, the controller performs these function calls sequentially, in the order specified. In the event-driven sequential mode, the application executes as the result of the executive servicing events that are asynchronously raised by the application and executive subsystems. Finally, in the time-driven sequential mode, the application executes as a result of the subsystems reacting to changes in the clock. This section provides a discussion of the non-event-driven sequential mode; Chapter III presents the other two modes of execution.

It was mentioned in Section 2.3.4 that the original version of Architect provided only rudimentary simulation capability, supporting only non-event-driven-sequential application

execution. In this mode, there is no application executive; each subsystem controller contains an update algorithm that determines the order in which the subsystem's primitive objects are updated. The application specialist specifies the update algorithm during the composition process. As Architect has evolved, this original execution mode has continued to be supported.

As an example, consider the simple application from the digital logic circuits domain shown in Figure 2.4. In this example, there are two switches (SW-1 and SW-2), an AND

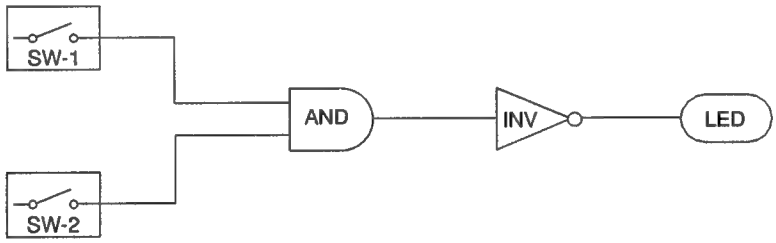

Figure 2.4 An Application from the Logic Circuits Domain

gate, an inverter (INV), and a light emitting diode (LED). Using AVSI, the application specialist creates a subsystem with these components. The corresponding AVSI representation of the subsystem and the technology base from which it was created are shown in Figure 2.5. The update algorithm for this subsystem would be as follows:

- Update SW-1
- Update SW-2
- Update AND
- Update INV
- Update LED

Prior to an object's Update function being called, there is no data in its output area. Therefore, it is the responsibility of the application specialist to select a sequence that

ensures no object is updated until all other objects supplying its inputs have already updated. In this example, the order in which the switches are updated could be reversed, and the the application would still demonstrate the correct behavior.

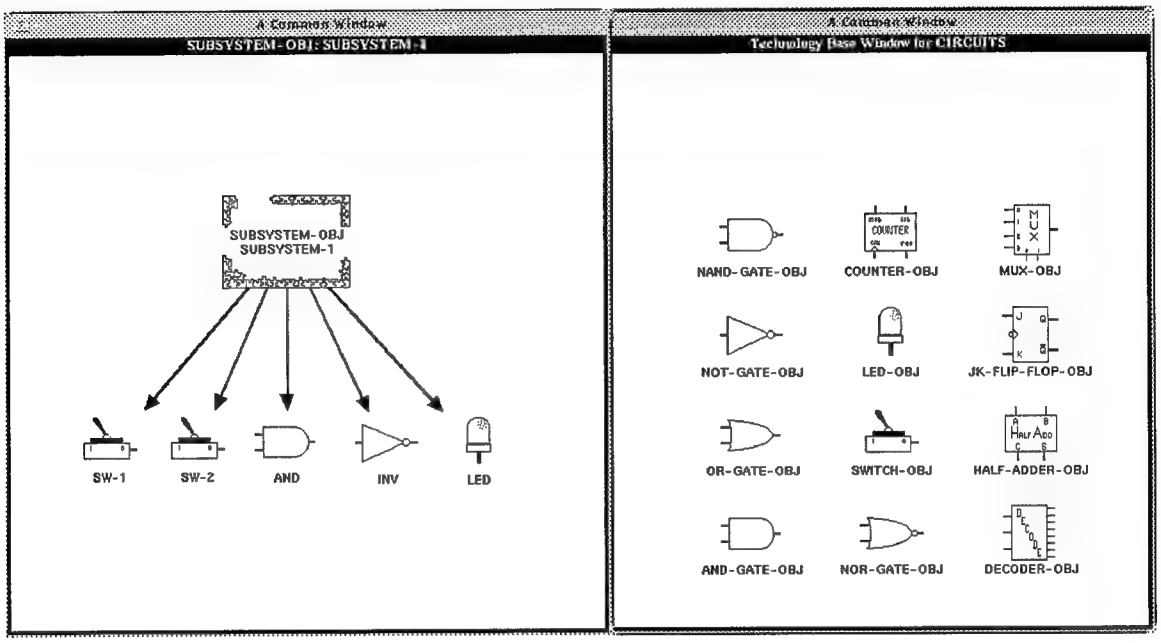

Figure 2.5 Architect Subsystem and Technology Base

This example also demonstrates how the knowledge base can be populated with new reusable models. The circuit designed in this example exhibits the behavior of a NAND gate. Once this behavior has been verified and the application saved to the technology base, the application specialist can use it in future applications. This is a simple example, however, creating increasingly complex models from lower level models allows the application specialist to reuse design knowledge from previous projects and to abstract out low-level details once the lower level models are instantiated. As an example, an application specialist designing a central processing unit (CPU) could use the lowest level primitives from the digital logic domain to design a half-adder. Two half-adders could, in turn, be used in

the design a full-adder which could, in turn, be used in the design of the arithmetic logic unit (ALU) which could, finally, be used in the design of the CPU.

2.4 Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL)

VHDL is a standardized and well-documented language. Additionally, it provides a simulation environment similar to what the application executive is attempting to achieve within the framework of the Architect system. Therefore, VHDL provided an excellent case study to analyze and compare the existing implementation of the application executive. This section provides a simplified description of VHDL. For a more detailed discussion of the VHDL language and simulation environment, see IEEE Std 1076-1987 (13), Lipsett (17), and Perry (20).

Interestingly, hardware engineers, as well as software engineers, have been searching for a tool to specify the behavior of their systems. In 1981, the DOD contracted for an industry standard, technology independent hardware description language which generates executable specifications. VHDL's goal is to describe systems at a number of different levels of abstraction and to simulate systems at any mixture of those levels (17, 20). Since the mid 1980s, engineers have used VHDL to specify the behavior and structure of their designs, and they have used VHDL's event-driven simulator to validate them.

In VHDL, operations are referred to as *processes.* Each process contains one or more statements that are executed sequentially. Each process defines a specific action or behavior to be performed when the value of one of the signals on its sensitivity channels changes.

A *design entity* is the basic VHDL unit of description. A design entity is used to represent individual components or functions which make-up a system. VHDL allows users to generate multiple copies of an entity by instantiating it one or more times in a system. A design entity has *ports* through which it communicates with the "world." A design entity consists of an entity declaration and an architecture body. The entity declaration defines the inputs and outputs of the entity so other components can interface with it. The architecture body describes the design entity in one of two ways:

- *structural description* – a composition of existing design entities
- *behavioral description* – a procedural description of the entity's transformation of inputs to outputs

A design entity can be described and decomposed using a hierarchy of structural definitions, but ultimately, lowest level entities must have behavioral descriptions.

Entities communicate via *signals.* Signals are like variables, except they are managed by *signal drivers.* A signal driver is like a queue; it holds the current value and all currently scheduled future values for the signal it is associated with. When a process generates a value on a data pathway, it may also designate the amount of time before the value is sent over the pathway. This is referred to as scheduling a transaction after the given time, and it is possible to schedule any number of transactions for each data pathway. The driver is a set of time/value pairs which hold the value of each transaction and the time at which the transaction should occur. The simulator updates signals during event driven *simulation cycles.*

VHDL has a two-stage model of time referred to as the simulation cycle. Figure 2.6 depicts the VHDL simulation cycle. The simulation cycle is the abstraction under which

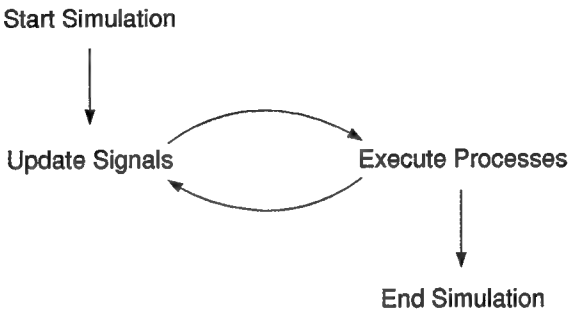

Figure 2.6 VHDL Simulation Cycle(17:12)

the hardware model described by the VHDL code is executed. This abstraction is based on a generalized model of the stimulus and response behavior of digital hardware (i.e. a functional component reacts to activity on its input connections and responds through its output connections).

During the first stage of the simulation cycle, the simulator updates all signals according to the values scheduled for the current simulation time in the signal driver schedules. This stage is complete when all data pathways which are scheduled to obtain new values at the current simulation time are updated. During the second stage of the simulation cycle, the simulator executes all processes that are sensitive to the updated signals. The execution stage will most likely schedule additional events in the signal drivers. When there are no more events scheduled for the current time, the simulation clock is set to the next simulation time at which a transaction is to occur and the cycle is started again.

During the execute process portion of the simulation cycle, processes are either active or suspended. A process is active when an event occurs on one of its sensitivity channels

during the current simulation cycle. A process is suspended when no events occur on its sensitivity channels during the current simulation cycle.

From the above model of the simulation cycle, it is evident that there is always some delay between the time a process puts a value on a data pathway and the time at which the data pathway sees that value. For example, if no delay is specified in an assignment statement of a value to a signal, VHDL uses what are known as *delta delays*. Delta delays result in repeated simulation cycles without the advancement of the simulation clock.

In VHDL, multiple processes can be active at the same simulation time. All processes that receive events on their sensitivity channels during the update signals part of the simulation cycle will be active during the execute processes part of the simulation cycle. This is one of the great advantages of VHDL – parallel event simulation. It also requires the designer to define the behavior the system at two levels: the sequential level and the concurrent level. The sequential level involves programming the behavior of each process which will be used in the model. The concurrent level involves defining the relationship between the processes, with particular attention to the communication between them (17).

2.5 Simulation

The application executive is the key OCU subsystem responsible for the simulation of an application in the Architect system and is the primary focus of this research. Therefore, it is the purpose of this section to provide a brief introduction and understanding of simulations, and in particular, to differentiate between time-driven simulation and event-driven simulation.

A simulation is the imitation of the operation of a real-world process or system. It involves the generation of an artificial history of a system, and the observation of that artificial history to draw inferences concerning the operating characteristics of the real system (4). The behavior of a system as it evolves over time is studied by developing a simulation model. This model usually takes the form of a set of assumptions concerning the operation of the system. Once developed and validated, a model can be used to investigate a wide variety of "what if" questions about the real-world system. Potential changes to the system can first be simulated in order to predict their impact on system performance. Simulation can also be used to study systems in the design stage, before such systems are built. Thus, simulation modeling can be used both as an analysis tool for predicting the effect of changes to existing systems, and as a design tool to predict the performance of new systems under varying sets of circumstances (4).

2.5.1 System Simulation. Since Architect provides a simulation environment for a system, it is important to define system simulation. System simulation is defined by Khoshnevis as "the practice of building models to represent existing real-world systems or hypothetical future systems and of experimenting with these models to explain system behavior, improve system performance, or design new systems with desirable performances" (15:2). Furthermore, a system can be classified as discrete or continuous according to the way the system changes state with respect to time. System state is determined by the value of all the state variables in the system. A discrete system is one in which all state variables have discrete values. The state of a discrete system changes in finite jumps or quanta according to the discrete values of its state variables. State transitions are caused

by stimuli called events. Events are associated with an instantaneous point in time and have no duration. A discrete system can be modeled as a finite state machine. A continuous system has state variables that can take on a continuous range of values. The state of a continuous system is characterized by smooth, continuous changes. Continuous systems are often represented by sets of differential equations that specify the system's behavior (10).

2.5.2 Event-Driven and Time-Driven Simulation. An important characteristic of simulations is the way in which the advancement of time is handled. In an event-driven simulation, events are raised asynchronously by components in the simulation. As the events are processed, the simulation clock is updated so that its time is the same as the time of the event currently being processed. The clock is, therefore, driven by the execution of the simulation model. In a time-driven (or time-stepping) simulation, the clock is advanced in uniform increments. At each increment, model components are given the opportunity to execute. The model components respond to changes in the clock but do not raise new events. After each component has been given the opportunity to execute, the clock is incremented, and the process is repeated (29).

2.6 Summary

The software architecture of Architect is based on the OCU model, and OCU is particularly useful in modelling software systems that can be described as a set of subsystems. In essence, the subsystems, or models, can be thought of as representations of real-world objects with mappings to real-world implementations. The relationships between objects

in the model must be consistent with the real world. Similarly, simulation of the model must be consistent with real-world implementation. It is the responsibility of the application executive to control the simulation and to ensure this consistency. For example, when the application executive provides the notion of time, it must associate simulation time with simulation behavior the way real-world time and real-world behavior are associated. With this in mind, it is evident that the application executive is the key element in the execution of applications within Architect, and is the focus of the remainder of this paper.

III. Architect's Application Executive

3.1 Introduction

This chapter provides a comprehensive description of Architect's simulation environment which is controlled by the application executive. The incorporation of the application executive into Architect was the result of three separate research efforts (9, 29, 32). As a result, the information detailing its implementation is spread throughout three thesis reports. This chapter brings all the information together into a single document and resolves any inconsistencies between the individual research efforts.

Chapter II introduced the three execution modes supported by Architect and described the non-event-driven sequential mode which is controlled by an update algorithm. The primary focus of this chapter is on the two modes controlled by the application executive: event-driven sequential and time-driven sequential. Both of these modes involve handling events and the passing of time. The major difference is in how the advancement of time is handled, as described in Section 2.5.2. In essence, an event-driven application executes as the result of the executive servicing events that are asynchronously raised by the application and executive subsystems. A time-driven application executes as a result of the subsystems reacting to changes in the clock.

The purpose of this chapter is to document the current implementation of Architect's simulation environment. It does not attempt to analyze or evaluate this implementation; it merely consolidates the available information into a single, comprehensible document. Chapter IV provides an analysis of this implementation and identifies areas requiring modification.

3.2 Big Picture

Before focusing strictly on the application executive, it is important to gain a "big picture" understanding of an event-driven application in Architect. Figure 3.1 shows an object model of event-driven applications (29:19). This model was created using the tech-

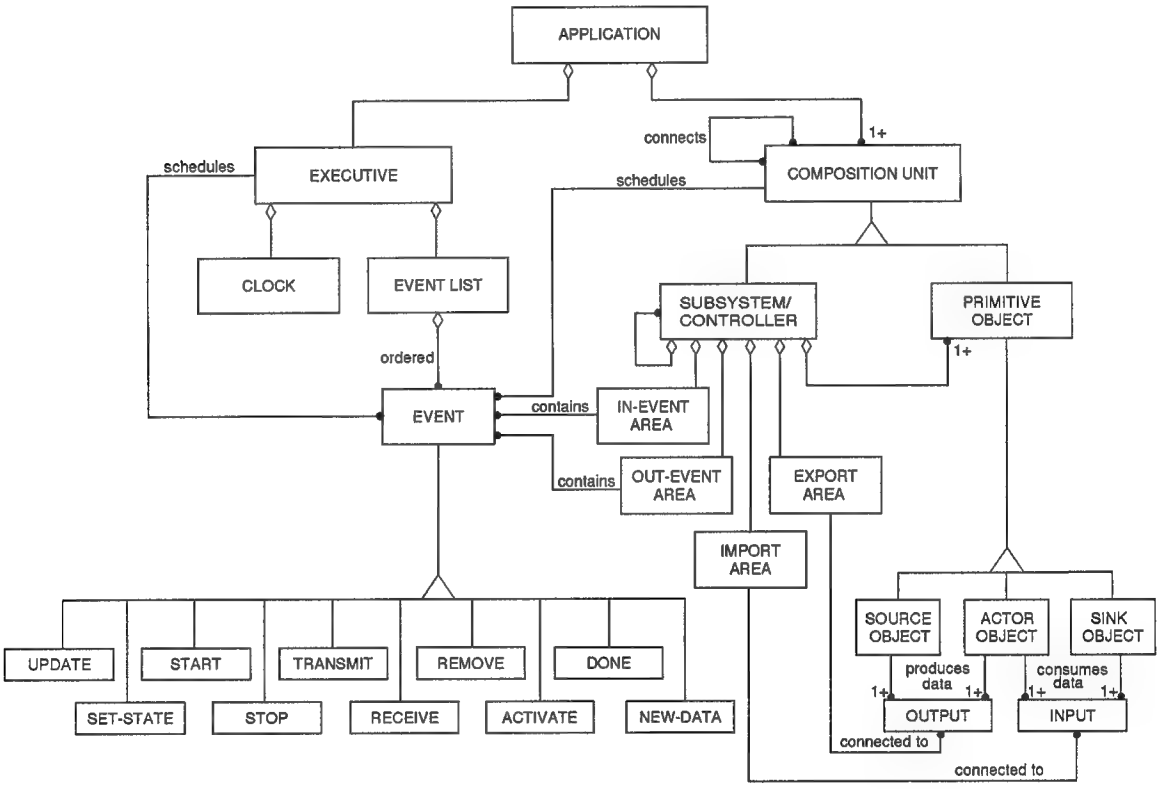

Figure 3.1 Object Model of Event-Driven Domain in Architect

niques of Rumbaugh as explained in (27). The model shows that an application is composed of one or more composition units and an executive. Each composition unit (abstract class) can connect to zero or many other composition units and is either a primitive object or a subsystem (concrete classes). Subsystems are in turn composed of other subsystems, primitive objects, an import area, and an export area. These parts describe the components

of the OCU model discussed in Chapter II. In order to support event-driven execution, several additions were incorporated into the basic OCU model. The executive needs to maintain a list of events, and it requires a clock to monitor the passage of time. Ten different types of events are used to control the execution of the application; these are defined later in Section 3.5.1. The events are sent to and collected from the InEvent and the OutEvent areas of top-level subsystems by the application executive.

3.3 Domain Analysis

The application executive is a supervisory program, and as such, it manages resources for the rest of its associated application. The domain analysis techniques of Prieto-Díaz (21, 22) and Tracz (28) were adapted to define a domain analysis process which was used to perform a domain analysis over the domain of supervisory programs. The result of this domain analysis was the creation of an object model of an application executive. A detailed discussion of the domain analysis process can be found in Appendix A of Welgan (32), and the resulting object model appears in Figure 3.2.

3.3.1 Application Executive Services. One of the key results of the domain analysis was the identification of services that an application executive must provide to the composed models (subsystems). These services, as defined by Welgan (32:24-25), are as follows:

1. *Event Handling* - The executive services events raised by the application during execution. The executive orders events and may generate events for the application executive and the application subsystems.

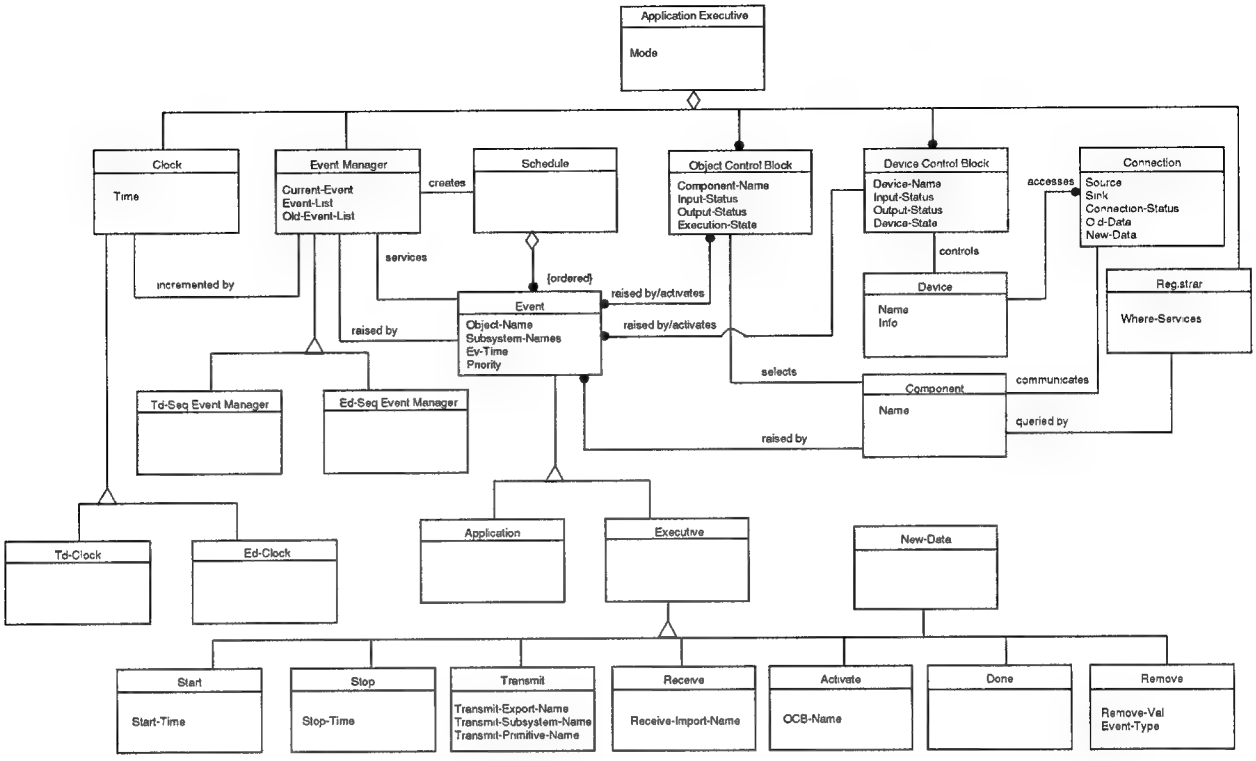

Figure 3.2 Application Executive Object Model(32:27)

2. *Registration* - The executive receives application-specific requests for executive services. This involves building executive data structures with information about application subsystems. This enables the executive to keep track of each subsystem's execution state. For example, the executive subsystem collects the connections in the composed application and controls them throughout application execution.

3. *Activation* - The executive activates particular subsystems when it's time for them to execute. The order of activation is determined by the schedule and by the current execution state of the component that must execute. Activation is the result of an event and may cause other events to be generated.

4. *Communication* - The executive supervises internal application data transfer. It manages connections between components and provides inputs for processes when it is time for them to execute.

5. *I/O Handling* - The executive supervises application calls for external I/O services. It controls external device interaction with the application. It manages application access to device drivers and buffers for those devices.

6. *Scheduling* - The executive makes an execution schedule for the application. It orders events and determines how to serialize concurrent events.

The application executive domain model shown in Figure 3.2 depicts the objects that were identified to carry out these services.

3.4 Transformation from Domain Model to Architect's OCU-Based Architecture

The application executive domain model consists of objects and operations that perform the executive services listed in Section 3.3.1. The OCU architecture provides a useful model in which to encapsulate these objects and operations.

3.4.1 Executive as a Single, Top-Level Subsystem. Currently, Architect's application executive consists of a single, top-level subsystem for each of the event-driven sequential and time-driven sequential modes. The subsystem controls a set of primitive objects that encapsulate the services required by the executive. The application executive primitives, as defined by Welgan (32:38-39), are as follows:

- Component Manager Primitive
- Device Manager Primitive

- Event Manager Primitive
- Connection Manager Primitive
- Clock Manager Primitive

The first two primitives are not currently used; they are included for future expansion to executives that support concurrent execution of application subsystems. The event manager primitive handles all events for the application. This primitive contains an update method that adds events to an event list, services the event, and removes the event from the event list. It employs Refine^{TM} function calls that pass control to application components and collect events raised by the components during execution. The connection manager primitive manages a set of connection objects that link subsystems in the application. This primitive contains update methods that return connection state when needed. The methods also read data from component export areas and write data to component import areas directly. The connection manager also raises events that must be placed on the event list by the event manager. The clock manager primitive keeps time for the application.

The domain model of Figure 3.2 contains more objects than the primitives listed above. For example, there is no registrar primitive. The registration of application components is carried out implicitly during the composition process, therefore, the registrar is not included explicitly as a primitive object. Similarly, the device object and the component object are part of the composed model (32).

3.5 Incorporation of Events and Event Processing

With the incorporation of the application executive into Architect, it was necessary to determine the type and structure of events that would be needed to support the new modes

of execution. In addition, there were several architectural modifications made during the incorporation of the application executive.

3.5.1 Events Types. One of the first steps in adding event processing to Architect was the identification of the types of events needed to control the execution of an application. Events were broken down into two classes: application events and executive events. All subsystems need to manage (route) both classes of events; distinction between the two lies in which type of subsystem does the actual processing of the event. The executive processes executive events, and the application subsystems process application events.

3.5.1.1 Application Events. There are two types of application events. The first is the *Update* event. This event is used to tell a primitive object to execute its Update function. The second is the *SetState* event. This event is used to establish a new state for a primitive by setting one or more of its attribute values and possibly making these values available as output.

3.5.1.2 Executive Events. Similarly, there is a set of Executive events. *Start* and *Stop* control the beginning and ending times of the simulation. *Transmit* and *Receive* control the exchange of data between top-level subsystems. *NewData* notifies a top-level subsystem it has new data in its import area. *Remove* notifies the event manager that an event has become stale and needs to be removed from the event list. *Activate* and *Done* events are included for future implementation of concurrent modes.

3.5.2 Event Structure. Once the event types were resolved, the structure of events was defined. It was determined that all events needed the following information:

the time the event was to take place, the path to the target (routing scheme), the priority of the event, and the target of the event. This common information is depicted in Figure 3.3, where the ellipsis implies that the event may contain additional information.

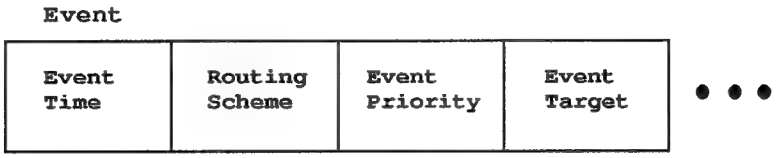

Figure 3.3 Common Information for All Events

The additional information varies from one type of event to another. For the Update and NewData events, no additional information is required. The SetState event contains additional, domain-specific information pertaining to attribute names and attribute values of the target primitive. The Receive event has an additional field denoting the target import object for the data being sent. The Transmit event has three additional fields: one identifying the export object the data is coming from and one each for the subsystem and primitive that produced the data. The Start event has a field for the start time, and, similarly, the Stop event has one telling the stop time. The Remove event contains information for the event manager, specifying the type and name of the event to be removed from the event list. The Activate and Done events have additional fields but are not expanded on here since they are not used in the sequential modes of execution.

3.5.3 Architectural Modifications. In order to support the application executive and event processing, it was necessary to make some modifications to Architect's OCU architecture. This section summarizes the changes and additions that were made; a complete description of the architectural changes can be found in (9).

3.5.3.1 Consolidation of Import and Export Areas. With the addition of the application executive, Architect needed to provide support for multiple independent subsystems. An independent subsystem is a hierarchical structure of subsystems and primitives that is a self-contained composition. It is unaware of any other subsystems and exhibits a complete behavior. Any composed application with an application executive has at least two independent subsystems, since, in the OCU sense, the executive is just another subsystem. These two subsystems are independent, communicating subsystems and neither is subordinate to the other. The incorporation of independent subsystems prompted a change in the original implementation of the import and export areas. The original implementation, in keeping with OCU documentation (16), had an import area and an export area associated with each subsystem of a composed application (see Figure 3.4). According to Gool, this was no longer feasible because "allowing multiple independent subsystems to connect to subordinate subsystems of other independent subsystems obligated subsystems to know more about their environment than intended by the SEI" (9:6-10). Therefore, the import areas and export areas were consolidated at the top-level subsystem of each independent subsystem (see Figure 3.5), and independent subsystems within an application can only communicate and share information through their import and export areas.

In order to support this change, it was necessary to ensure that only the appropriate primitive within an independent subsystem could access its import and export data. This was accomplished by partitioning the import and export areas and adding the owning subsystem and the import/export path to data in the import/export areas. To illustrate this, Figure 3.6 shows a sample independent subsystem, and Figure 3.7 shows what data

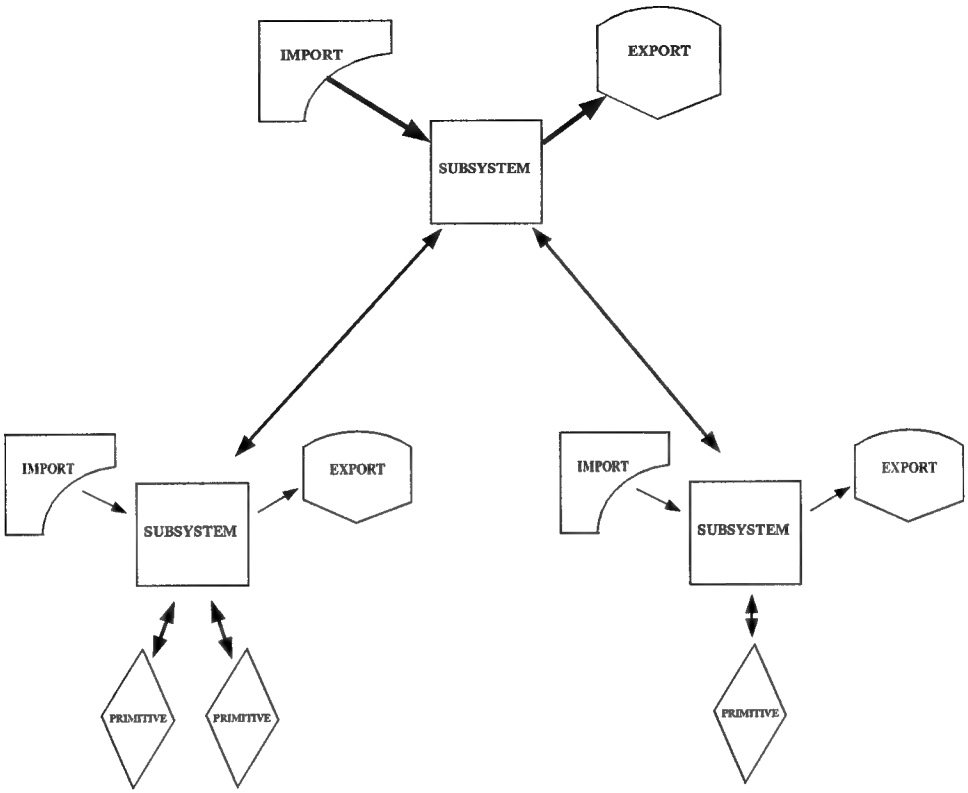

Figure 3.4 Original Import Export Implementation

in the associated import and export areas might look like. In Architect, the data takes the form of import and export objects. The data fields shown in Figure 3.7 are attributes of the import and export objects.

The top-level subsystem is now responsible for managing access to the import area and export area for all of its subordinate primitives. When the import area is changed as a result of a connection object placing new data there, the executive informs the top level subsystem of the new data via a NewData event. If the new data is for one of the top-level subsystem's primitives, it schedules an Update event for that primitive. Otherwise, it passes the NewData event down the hierarchy to the appropriate subordinate subsystem

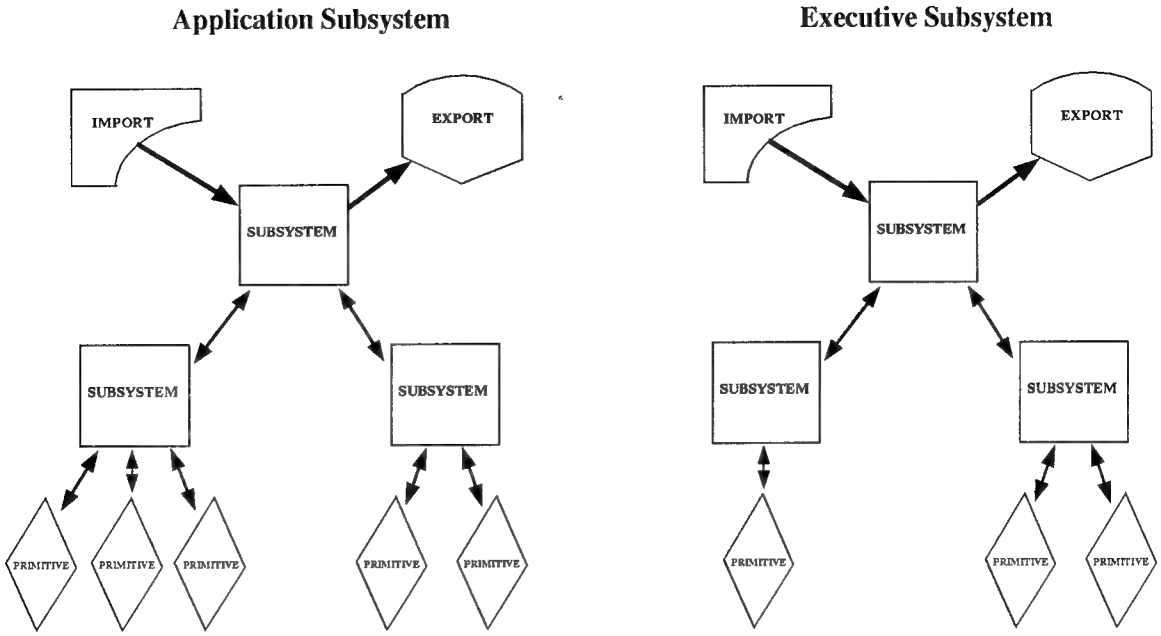

Figure 3.5 Independent Subsystems Represented as an Executive Subsystem and an Application Subsystem

that is superior to the primitive identified as the destination of the new data. The details of the flow of events through subsystems can be found in Section 3.5.4.

3.5.3.2 Connection Objects. Another by-product of introducing an application executive and independent subsystems into Architect was the need to modify the way that subsystems communicate. Imports and exports within an independent subsystem communicate via *source* objects and *target* objects. These two types of objects are functional converses, and they identify either the source export object that produces the information for this import object or the target import object for the information produced by this export object.

In addition, there needed to be a way for independent subsystems to exchange data. A *connection* object was developed to perform this function. The connection object serves

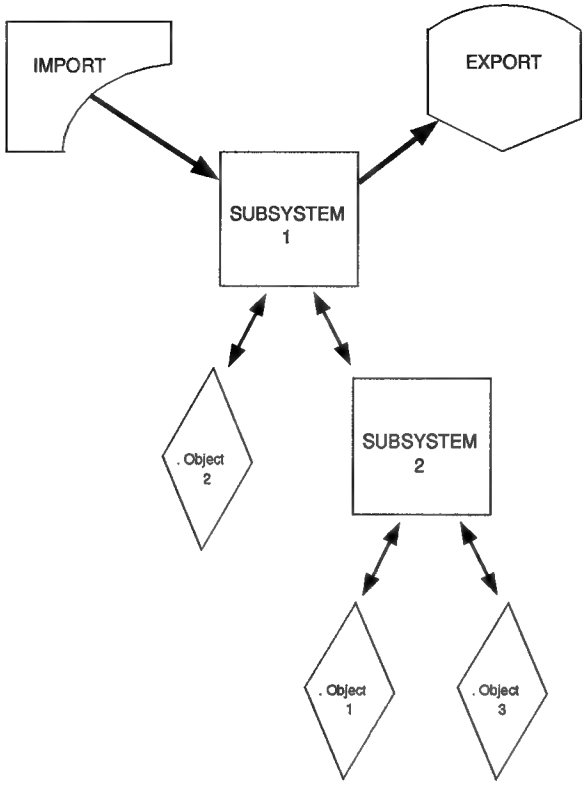

Figure 3.6 Sample Subsystem

the same purpose as the source and target objects; however, it connects the import and export objects of independent subsystems. The connection object serves as a link between independent subsystems that buffers the data that flows between them. It has associated with it a source object (export object) and a sink object (import object). The connection object raises a Receive event when it gets new data. This notifies its sink object that it has new data which must be considered by the sink object as it executes.

Figure 3.8 shows the relationship between source, target, and connection objects. Keep in mind, each of these connect import and export *objects,* not import and export *areas.* Each import (export) area consists of a *set* of import (export) objects, and there is an import (export) object for each separate data item in the import (export) area.

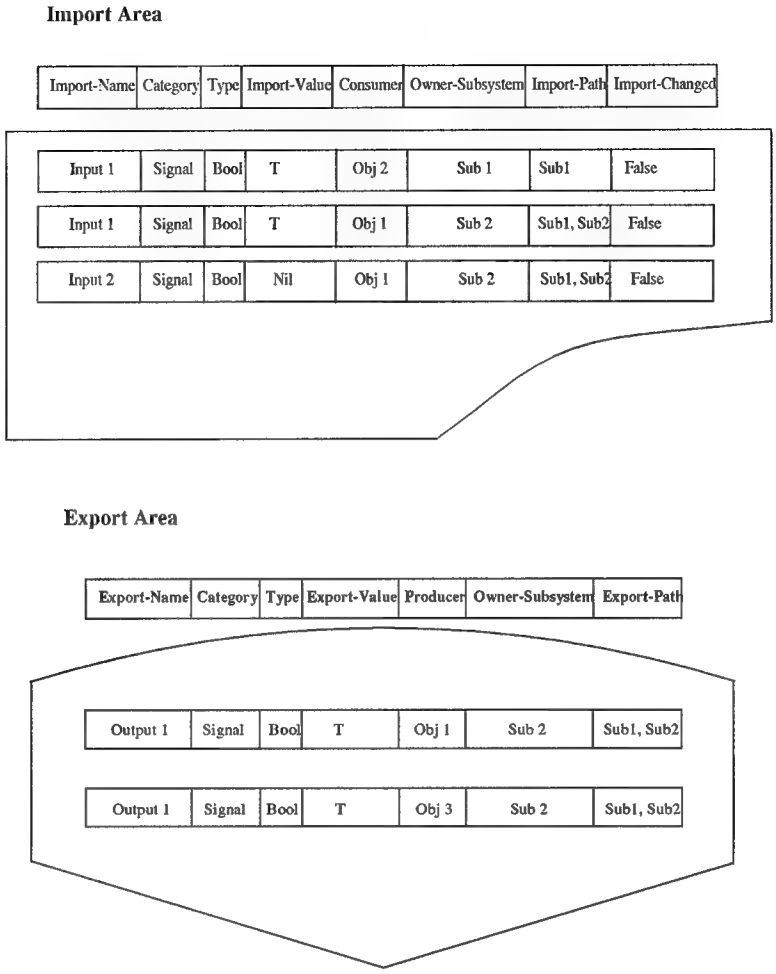

Figure 3.7 Sample Import/Export Information and Data

Furthermore, each export object may need to provide its information to more than one import object, resulting in more than one connection object associated with each export object. Therefore, each import (export) area has potentially many connection and/or source/target object pairs associated with it. Figure 3.8 does not show all the source, target, and connection objects that would be instantiated for this application. It displays only one of each type in order to show the relative location of each in a composed application.

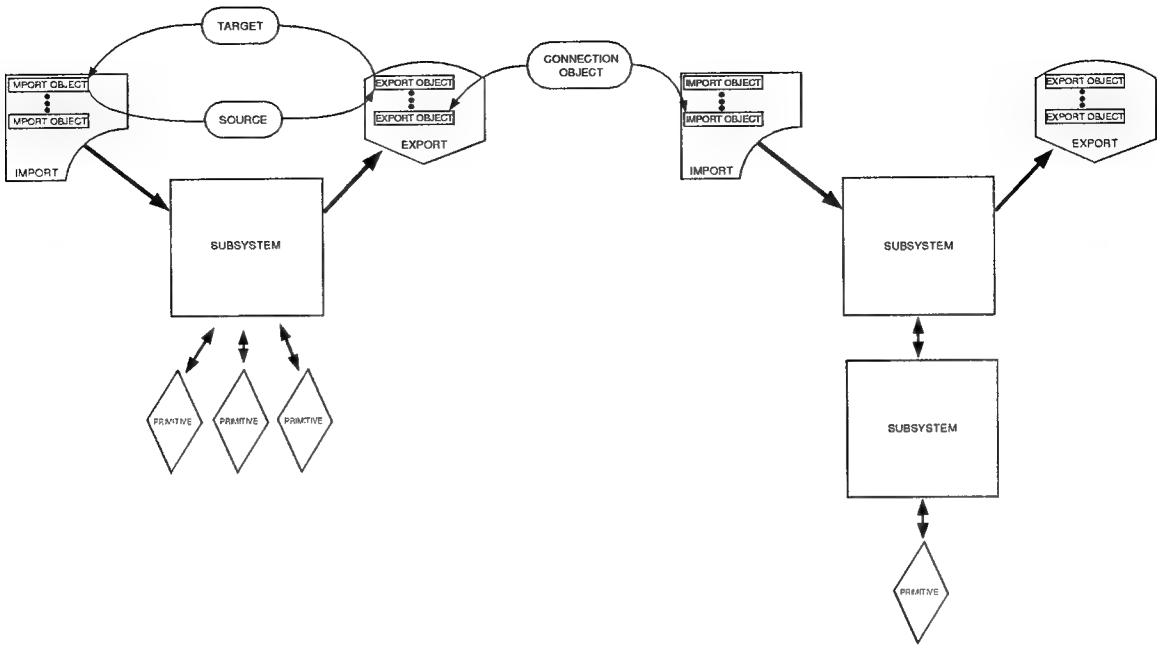

Figure 3.8 Independent Subsystems with Source, Target, and Connection Objects

3.5.3.3 InEvent and OutEvent Areas. The previous section discussed how data moves between independent subsystems in Architect by using import and export areas of top level subsystems. The import area and export area are provided by Lee as components of the OCU model (16); however, he does not make any reference to events or event processing within the OCU framework. The implementors of Architect chose to keep event flow separate from data flow. It was decided not to incorporate events into the import and export areas but to provide separate InEvent and OutEvent areas. With these constructs, event passing was implemented in much the same manner as data passing. Events flow into the subsystem via the InEvent area, and events exit the subsystem via the OutEvent area. Although similar to the import and export areas, one significant difference exists: *all* subsystems have InEvent and OutEvent areas where only the *top-level*

subsystems of independent subsystems have an import area and an export area. Figure 3.9 shows an example subsystem with InEvent and OutEvent areas.

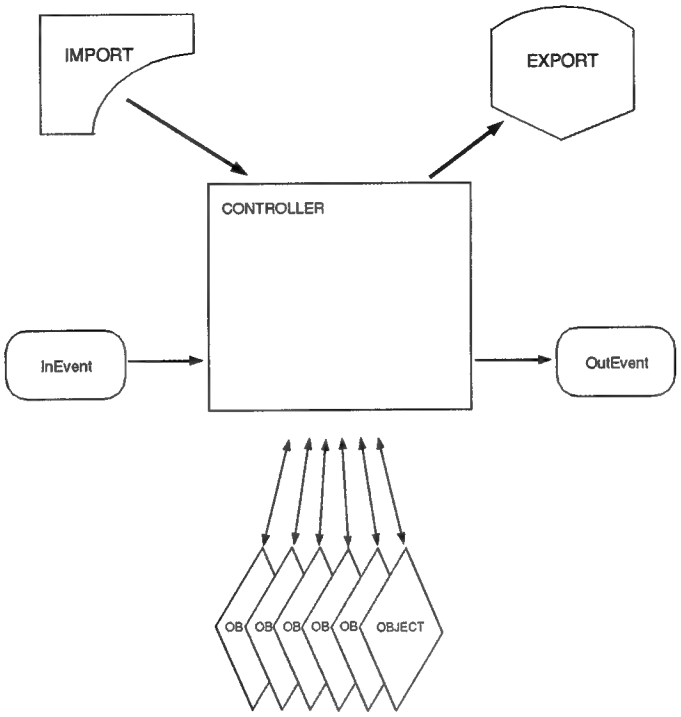

Figure 3.9 Subsystem With InEvent and OutEvent Areas

3.5.4 Flow of Events Through Subsystems. As with data flow, events enter independent subsystems via the top-level subsystem. When the executive processes an event, it determines which top-level subsystem the event's target primitive is subordinate to, places the event in that subsystem's InEvent area, and invokes the subsystem's Update function. When the top-level subsystem is updated, it retrieves the event in its InEvent area, interrogates the event, and uses information encapsulated in the event to decide what to do next. The information required is in the routing scheme; the routing scheme of the event contains a sequence of subsystems that are superior to the target object of the

event. Figure 3.10 shows an example subsystem with an event targeted for a subordinate primitive; emphasized are the target primitive and routing scheme fields of the event.

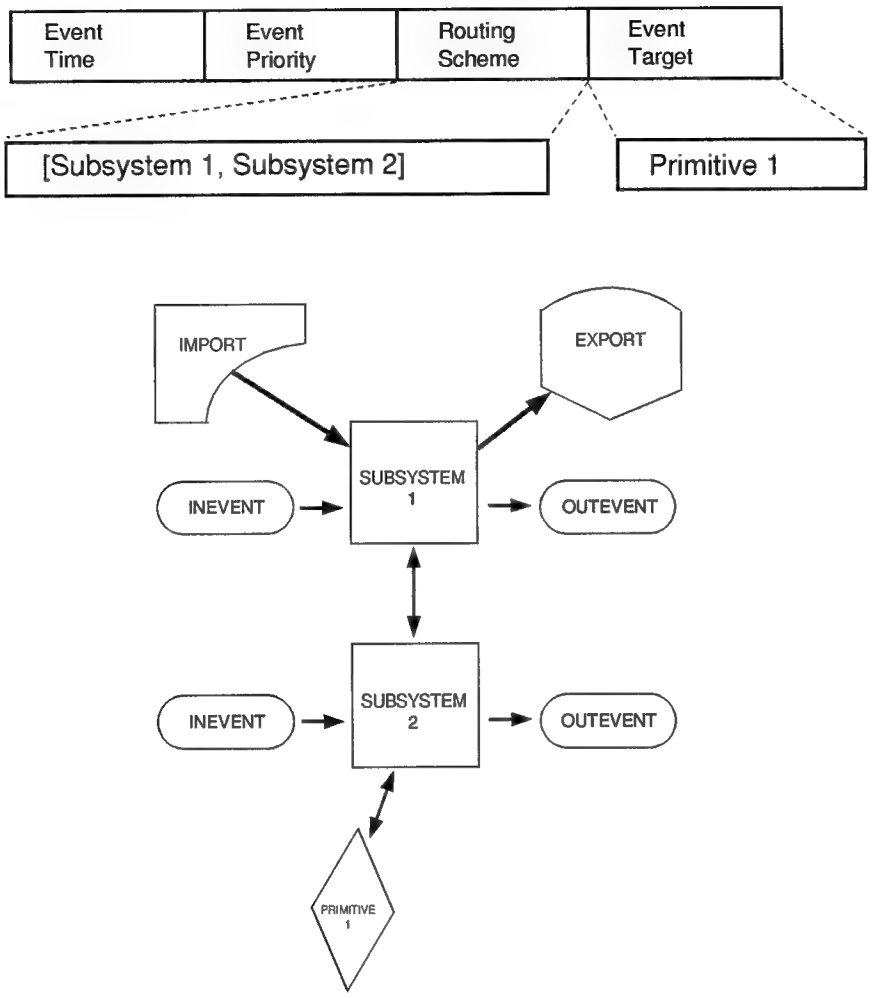

Figure 3.10 Example Subsystem with Associated Event Targeted For Primitive

Using the event's routing scheme to process the event, the top-level subsystem will remove itself as the first element of the sequence. If there are any subsystems remaining in the routing scheme, the top-level subsystem passes the event to the InEvent area of the subsystem that is now at the head of the sequence and invokes that subsystem's Update function. The event is passed down the independent subsystem's hierarchy in this manner

until, eventually, the sequence of subsystems in the routing scheme is empty. When this occurs, the destination primitive is subordinate to the current subsystem. The subsystem consumes the event and invokes the appropriate function of the target primitive. For example, if the subsystem consumes an Update event for one of its primitives, the controller of the subsystem will invoke the Update function of the associated primitive. When the primitive finishes its operation, any newly generated application events are passed back up the hierarchy of the independent subsystem, creating the routing scheme for the newly generated events. When the events reach the top-level subsystem, they are passed to the event manager of the application executive for insertion into the event list.

3.5.5 Subsystem Controller. In Section 2.3.5, the update algorithm for the non-event-driven sequential mode of operation was introduced. In this mode, the update algorithm serves as the Update function for the controller of the subsystem. Each subsystem requires an update algorithm, furthermore, each update algorithm must be individually specified by the application composer. In the event-driven and time-driven modes, all subsystem controllers in the application have identical Update functions. When called by the executive, the Update function gets an event from its InEvent area and either routes or consumes the event as described in Section 3.5.4. Once the event has been processed (either by this subsystem or a subordinate subsystem), any new events raised during the processing of this event are passed from the OutEvent area to the application executive for insertion into the event list. Finally, before relinquishing control, the controller clears its InEvent and OutEvent areas. Having identical Update functions simplified the design of the controller and simplifies the composition process for the application composer.

3.5.6 Event Servicing. The event manager primitive maintains an event list which is a sequence of events. The ordering of the events in the sequence is determined first by their time and then by their priority. When the executive subsystem calls the Update function of the event manager primitive, it services the next event in the event list. The servicing of an event may include starting/stopping execution, adding/deleting events to/from the event list, or routing events to application subsystems. In event-driven sequential mode, when an event is serviced, it is removed from the event list and saved in an "old events list". This list is available to the user following application execution, thus allowing the user to view the actual sequence of events raised during execution.

3.5.7 Delay Simulation. In order to provide realistic simulation, many application domains require the environment to simulate delays associated with transient behavior of components. For example, consider the delay required for a signal to propagate from the input to the output of a component in the logic circuits domain. During this propagation, the component is in a transient state. The new output of this component should not be provided to any other components that use this signal as an input until the associated propagation delay has expired.

In the event-driven mode, the application executive uses the application events (SetState and Update) to control when a state changes with respect to time. To help demonstrate this, Figure 3.11 shows a simplified functional model (see Rumbaugh (27)) of a primitive object from an arbitrary event-driven domain and its interfaces with the architecture/executive. In accordance with the OCU model, the primitive object has Update and SetState functions and a set of attributes.

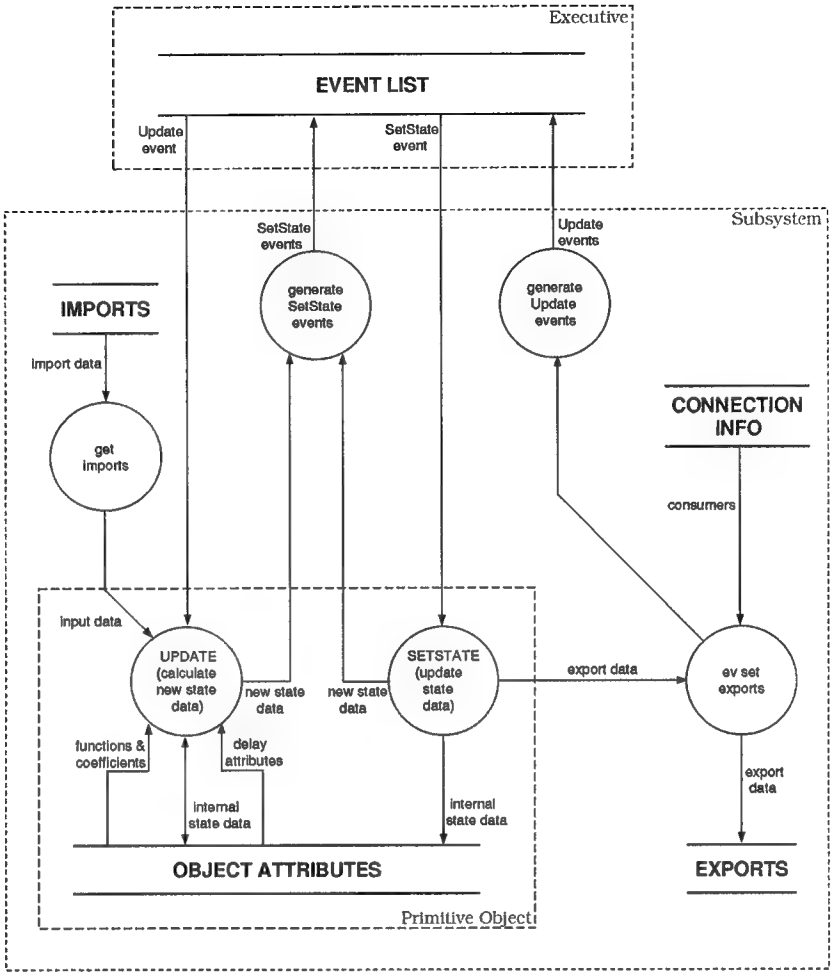

Figure 3.11 Simplified Event-Driven Functional Model

When an application subsystem receives an Update event for one of its primitives, it calls the Update function of the appropriate primitive. The Update function, in turn, invokes the architecture function *get-import* to retrieve new input data from the import area. The primitive uses this new data and its internal state data to calculate a new state. It then determines the delay associated with this new state and invokes the *gen-set-state-event* function to generate a SetState event for itself at the proper relative time. The SetState event is passed to the executive and is inserted into the event list. After

the delay has expired, the SetState event is serviced by the executive, and the object's SetState function is called. The SetState function updates the object's internal state and invokes the *ev-set-exports* function to update the object's export data. The new state data is now available to all other components that use it as input.

In addition to updating the object's external state via the export area, the *ev-set-exports* function determines (via connection information) which other objects in the application consume the export data, writes the new data to their top-level subsystem's import area, and invokes the *gen-update-event* function to generate Update events for each recipient of the data. These Update events are sent to the application executive for insertion into the event list.

3.6 Simulation Clock and Advancement of Time

Before continuing with the instantiation of the application executives, it is important to explain how the advancement of time is handled by the application executive. During application composition, the application composer selects an execution mode (see Section 3.8.1). Whenever a mode other than non-event-driven sequential is selected, Architect automatically creates a global simulation clock. The global clock is used by the application executive to keep absolute simulation time. Individual subsystems do not read the clock during execution but are provided the notion of time in a manner dependent on the chosen mode of execution.

3.6.1 Event-Driven Mode. In the event-driven sequential mode, subsystems raise events, stamp them with a relative time, and send them to the event manager for insertion

into the event list. The event manager keeps a copy of the current time in its import area, and when it receives an event to be added to the event list, it stamps the event with the correct absolute time before storing it in the events list. The global clock can only be updated to a time equal to $t + \tau$ (where τ is the relative difference in time between t and the scheduled time of the next event in the event manager's event list) when the following three conditions are true:

1. There are no more time t events scheduled.
2. It is not possible for any top-level subsystem to schedule anymore time t events.
3. The next event in the application occurs at τ units after the current time (32:31).

3.6.2 Time-Driven Mode. The advancement of time and management of events is handled differently in the time-driven mode. In the event-driven mode, primitives do not know about the passage of time except through the receipt of SetState events. In the time-driven mode, time is provided via a connection object to each primitive that requested it during registration (composition); therefore, primitives do not schedule SetState events for themselves. Since they receive time as an input, they will know when it is time to make their state data available to their consumers.

In this mode, the event list contains an Update event for each primitive that receives the simulation time and an Update event for the global clock. Following a Start event, the event manager services the application Update events in a round-robin fashion. When their Update function is called, the primitives react to the current time by either executing or not executing at that time. Following event servicing, the Update events are not removed from the event list unless explicitly told to do so by the application via a Remove event.

However, as in the event-driven mode, any executive events (i.e. Transmit or Receive) serviced during this simulation time are removed following event service.

Once all of the application Update events are serviced, the Update for the clock is serviced resulting in the global clock being incremented, thus simulating a clock tick. Following the "tick" of the clock, the event manager will repeat this cycle of round-robin activation of the application primitives until the clock reaches the stop time specified in the Stop event.

Figure 3.12 shows a simplified functional model of a primitive object from a time-driven domain. As with the event-driven mode (see Section 3.5.7), the primitive has an Update function, a SetState function, and a set of attributes. When an Update event for this primitive is serviced by the executive, the primitive's Update function is called. When it is updated, the primitive calls the *get-import* function, reads the time (placed there previously by the executive), and decides whether or not to execute depending on the current simulation time. If it is time for the primitive to execute, it retrieves new data from the import area using the *get-import* function and uses this data, along with its current internal state data, to calculate a new state. The primitive updates its internal state and calls the *ev-set-exports* function to update its export data.

3.7 Instantiation of Executives

Section 3.4.1 introduced the application executive as a single, top-level subsystem and developed the primitive objects needed to provide the services for the executive domain. The primitives themselves are not an executive until they are joined together under the

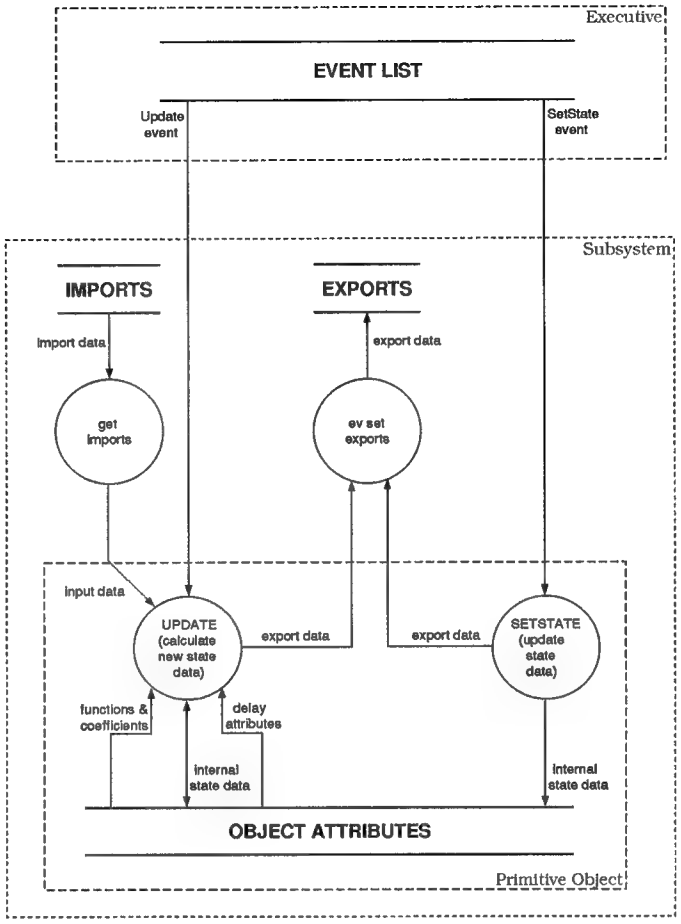

Figure 3.12 Simplified Time-Driven Functional Model

control of a subsystem controller. This section presents the subset of primitives used to instantiate the event-driven sequential and time driven sequential application executives.

In Architect, executable applications are expressed in a domain-specific language (DSL) that requires three grammars: a domain-specific grammar, an OCU grammar, and an executive grammar. The DSL needs to inherit from both the OCU and executive grammars. However, Dialect restricts a grammar to inherit from at most one other grammar. Therefore, the executive and OCU grammars were combined into a single grammar. The event-driven and time-driven application executives were then expressed using this com-

bined OCU/executive grammar and saved to separate files. Now, when the application composer selects the execution mode during application composition, the appropriate application executive subsystem is parsed into the application. With this implementation, the application executives do not need to be instantiated each time a new application is composed; the application executive subsystem is brought into the current application automatically and is transparent to the user.

3.7.1 Event-Driven Sequential Executive. The application executive subsystem for this mode consists of the following primitives:

- Event Manager Primitive
- Connection Manager Primitive
- Clock Manager Primitive

These primitives are sufficient due to the restrictions that sequential execution places on the application. The component manager primitive is responsible for keeping track of the execution state of all application subsystems. With a single thread of control, only one subsystem can execute at a time; the flow of control is passed from one subsystem to another and there is no need to manage the execution state of all subsystems. The device manager primitive manages the execution state of all input/output devices used by the application. Once again, with only a single thread of control, only one device can be active at a time, therefore, this primitive is not needed. In this mode, the instantiated executive already contains built in Start and Stop events. The application composer can change the associated start and stop times if desired. In addition, the application composer must insert at least one Update event into the event list to begin execution of the application.

3.7.2 Time-Driven Sequential Executive. The application executive subsystem for the time-driven sequential mode contains the same primitives as the event-driven executive defined in Section 3.7.1. This mode also has a single thread of control, and for the same reasons as above, does not require the component manager and the device manager primitives. As was the case with the event-driven mode, there are Start and Stop events built into the instantiated executive. Prior to execution of the application, the application composer must schedule Update events for all primitives that requested the simulation time during composition(see Section 3.6.2).

3.8 Executive Operation

The previous sections of this chapter developed the incorporation of the application executive into the current implementation of Architect. This section builds on the information developed previously to provide a high-level description of the application executive's operation. The executive has two phases of operation: registration and execution. These phases are presented below along with a brief discussion of the composition of an application.

3.8.1 Registration. During the registration phase, the application executive determines which components need executive services during the execution of the application. It was mentioned in Section 3.4.1 that registration is carried out implicitly during the composition of the application. With this in mind, the development of the registration phase begins with a look at the application composition process.

While composing an application using the AVSI, one of the first steps the application composer does is select the execution mode. Originally, this was accomplished by the AVSI prompting the composer for the desired execution mode, followed by a prompt for the domain (event-driven circuits, cruise missile, etc.) in which the application would be composed. There was no error checking to prevent an inexperienced user from choosing incompatible domains and executives. As part of a concurrent research effort, Guinto's (11) enhanced AVSI has fixed this problem. Instead of selecting the application domain and execution mode in separate steps, the application composer selects the domain and the appropriate application executive file is loaded automatically.

Once the execution mode has been selected, the application is composed using the mouse to drag and drop icons from the technology base. The icons represent the primitives from the domain-specific technology base, and in selecting an icon from the technology base, an instance of that primitive is created. The application composer creates the appropriate hierarchy of subsystems and primitives to specify the application. Next, the composer uses the AVSI to connect the inputs and outputs of the primitives. This step creates the connection, target, and source objects that connect the import and export objects. The final step in the registration phase is to schedule the initial set of events on the event list necessary to begin the execution, as presented in Section 3.6.

3.8.2 Execution. Following the registration phase, the application composer can choose to execute the application. The execution phase begins with the event manager servicing a Start event. Following the Start event, the execution continues in a manner dependent on the application executive mode selected. In the event-driven mode, the event

manager orders the events it services first by time and second by priority. In this mode, when the event manager services an event it is removed from the event list. In addition, the event manager must be able to add application (i.e., SetState) and executive events (i.e., Transmit) to the event list. Execution stops when the Stop event is serviced or the simulation time reaches the time specified by the attribute of the Stop event.

In time-driven mode, the current time is broadcast to all the primitives, and they react to the new time either by executing or not executing at that time. This is explained in more detail in Section 3.6.2.

3.9 Summary

This chapter presented a comprehensive description of Architect's application executive from domain analysis through implementation. The current executive consists of a single subsystem that provides executive services for application components. Two modes of execution are currently supported: event-driven sequential and time-driven sequential. The incorporation of the application executive into Architect required the evolution of its OCU-based software architecture to support event processing. Chapter IV provides a critical evaluation of Architect's simulation environment which is controlled by the application executive and identifies areas in which it can be improved and/or enhanced.

IV. Evaluation of Architect's Application Executive

4.1 Introduction

Chapter III provided a thorough description of the current implementation of Architect's simulation environment. This chapter analyzes the current implementation and identifies areas in which Architect's next generation simulation environment can be improved and/or enhanced.

In addition to the theory supplied by the applicable thesis reports (9, 29, 30, 32), the source code of the Architect system was reviewed to help analyze the application executive. The code helped clarify any inconsistencies in the thesis reports and helped identify ways in which the actual implementation differed from the intended implementation.

4.2 Source, Target, and Connection Objects

Section 3.5.3.2 presented how subsystems within an application communicate via source, target, and connection objects. There is no reason why two subsystems within an independent subsystem should exchange data any differently than two independent subsystems. This section presents a problem with the way data is currently exchanged and provides a solution that not only solves this problem but also standardizes the exchange of data between all subsystems in the application.

4.2.1 Violation of OCU's Visibility Rule. The current implementation of the source and target objects violates the visibility rule defined by the designers of the OCU model. According to SEI documentation, subsystems "...have no knowledge of other subsystems in the application" (16:18). Currently, each import object has an attribute

called *Sources*. This attribute is a map from the import object to a set of source objects. Each source object contains the name of a source subsystem and a source primitive that can provide this import object with its data. Through this Sources attribute, the import object can "look" to each of its potential sources for data. Similarly, each export object has a *Targets* attribute that is a map from the export object to a set of target objects. The target objects provide the export object with the target subsystems and target primitives of the data it is producing.

As an example of how this information is used, consider Figure 4.1. Except for the

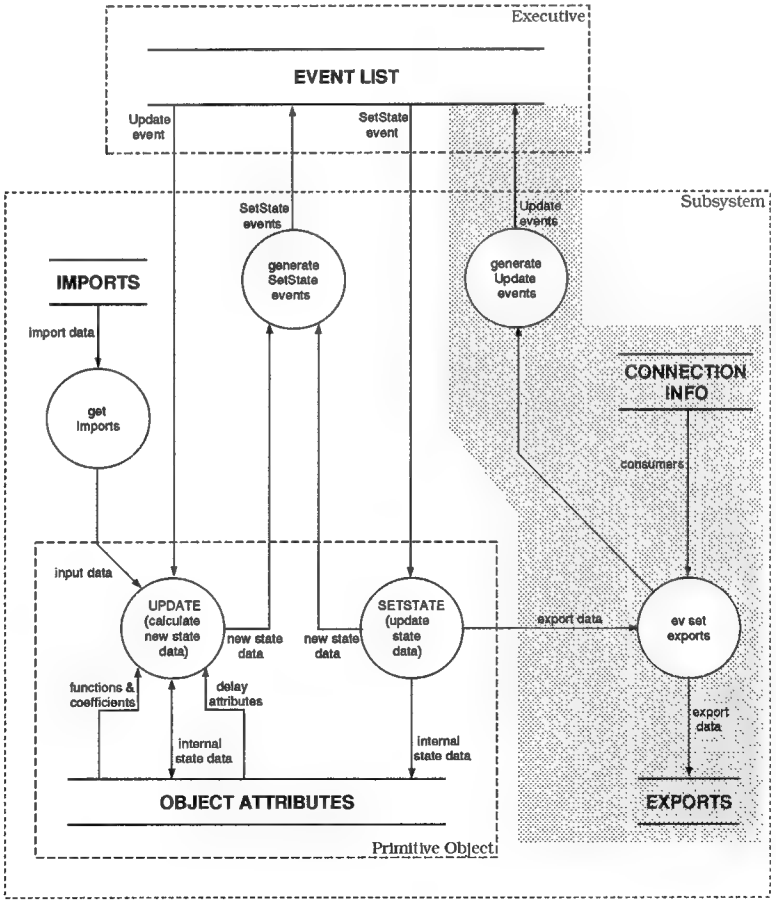

Figure 4.1 Event-Driven Functional Model with Highlighted Problem Area

additional shaded area, this diagram is the same simplified functional model introduced in Section 3.5.7. The shaded portion emphasizes the area where the visibility rule is violated. In this diagram, the primitive calls the *ev-set-export* function to update its external state. The algorithm of the *ev-set-export* function is as follows:

1. store the new data to the associated export object
2. use connection information to
   (a) store the data directly to all import objects that receive it
   (b) generate Update events for all target primitives of the data
3. generate a Transmit event to inform the connection manager that one of its connection objects needs to transfer data

The problem is the connection information originates from *within* the subsystem. It comes from the Targets attribute of the export object. The *ev-set-export* function enumerates over the set of Targets for this export object, storing the new data directly into the associated import objects in the target subsystems' import areas and generating Update events for the target primitives. This is clearly a violation of SEI's intent that subsystems do not have any knowledge of the other subsystems in the application.

4.2.2 Solution. In solving the visibility problem, it is also desirable to provide a standardized method to exchange data between all application subsystems. Due to the constructs already built into Architect, the solution to this problem does not require major modifications to the current implementation. The key is to create a connection object for each export object, not just for export objects connecting independent subsystems. Once each export object has an associated connection object, the *ev-set-export* function only

needs to update the data of the export object and then generate a Transmit event for the associated connection object. The details of incorporating this change are developed in the next section.

4.2.3 Implementation. Before specifically addressing the implementation issues, it is important to develop some background on how import, export, and connection objects are instantiated. Import and export objects are instantiated during application composition. When the application composer adds a primitive object to the application, the correct number of import and export objects are created, according to the number of inputs and outputs the primitive has. In addition, when inputs and outputs are connected, methods exist to create instantiations of connection objects and initialize the *Source-Exp* and *Sink-Imp* attributes of the connection objects. The Source-Exp attribute is a mapping from the connection object to an export object, and the Sink-Imp attribute is a mapping from the connection object to an import object. Currently, connection objects are only created for connections between subsystems of independent subsystems.

The first step in implementing the solution is to modify the Sink-Imp attribute to be a mapping from a connection object to a *set* of import objects. As connections are made, the Sink-Imp set can be constructed in the same manner that the Targets set is currently built. Next, the existing methods to create connection objects can be modified to enable instantiations between all subsystems not just top-level subsystems. When connections are made between primitive objects, if the source primitive's corresponding export object already has an associated connection object, the sink primitive's import object is added to the Sink-Imp set, otherwise, a new connection object is instantiated.

With this change, the algorithm for the *ev-set-export* function is simplified. The new algorithm is as follows:

1. store the new data to the associated export object
2. generate a Transmit event to inform the connection manager that one of its connection objects needs to transfer data

The corresponding functional model for this modification is shown in Figure 4.2. The

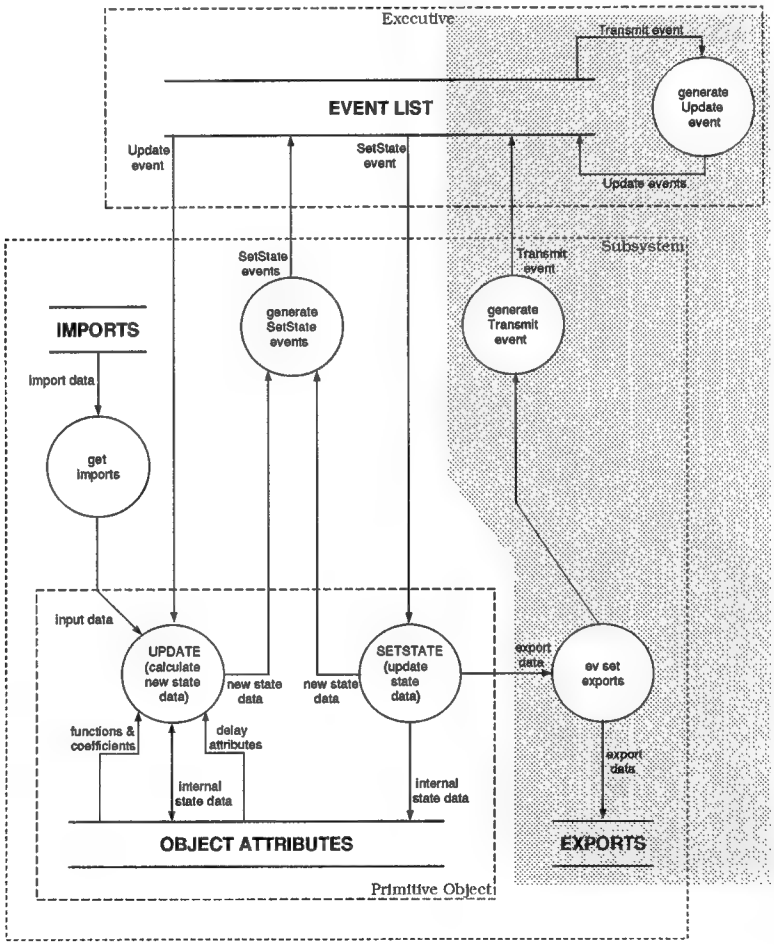

Figure 4.2 Modified Event-Driven Functional Model

shaded area shows the part of the model that has changed. Now when the primitive calls *ev-set-export*, the external state is updated and a Transmit event is generated for the

connection manager. When the Transmit event is serviced by the application executive, the new data is passed to each import object in the Sink-Imp set and an Update event is generated for each target primitive of the new data.

With this modification, each subsystem in the application no longer has any knowledge of the other subsystems. It only needs to generate Transmit events for the connection manager. When the Transmit events are serviced, the corresponding connection objects contain the information necessary to move the data between the subsystems in the application. Although this discussion concentrated on the event-driven mode, it applies to the time-driven mode as well, with one minor modification. In the time-driven mode, servicing of the Transmit event will only send the new data to the appropriate import objects; it will not generate any Update events.

4.3 Consolidation of Import and Export Areas

4.3.1 Deviation from OCU Standard. Section 3.5.3.1 explained how the import and export areas of an independent subsystem were consolidated at the top-level subsystems. The reason given for this deviation from the standard OCU model was that allowing subsystems from different independent subsystems to connect to each other violated OCU visibility restrictions. It is not clear how allowing these connections would give the subsystems visibility to each other. Assuming that it does, it is also not clear why one property of the OCU model (visibility) was preserved while another (all subsystems have import and export areas) was not.

4.3.2 Solution. Since insufficient justification could be found for consolidating the import and export areas at the top-level subsystems, the solution to this is to simply allow all subsystems to once again have import and export areas. The necessary changes are discussed in the next section.

4.3.3 Implementation. The first change is to modify the information encapsulated in each piece of data (import/export object) in the import and export areas. Figure 3.7 displayed sample import and export areas along with their associated import and export objects for a sample subsystem (Figure 3.6). The owning subsystem and the import/export path are no longer necessary. Figure 4.3 shows how the import and export objects are simplified with this modification.

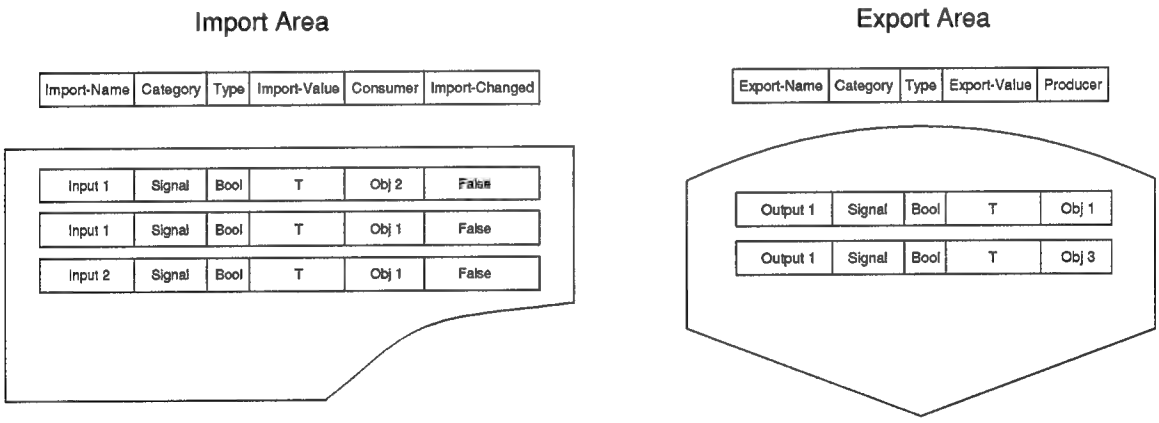

Figure 4.3 Modified Import and Export Data

Next, the code that creates import/export objects in the import/export areas needs to be modified. The code responsible for this is the *build-import-export-area* function. Once a subsystem is composed, this function is called by the visual system and has as an input parameter, a subsystem. It enumerates over the set of primitive objects that the input

subsystem controls and adds new import/export objects to the import/export area of the highest-level subsystem to which the input subsystem belongs. An import/export object is added for each input/output associated with the primitives.

The current implementation does not alter how subsystem objects are modelled in Architect. All subsystem objects still have an import area and an export area; however, only those belonging to top-level subsystems are currently utilized. Therefore, to support import and export areas for all subsystems, the only modification required for the *build-import-export-area* function is to have it add the import/export objects to the input subsystem's import/export areas instead of those associated with its the top-level subsystem. This change actually simplifies the *build-import-export-area* function since it no longer needs to identify the top-level subsystem to which the input subsystem belongs.

The modification of connection objects as developed in Section 4.2 supports the exchange of data between import and export areas of subsystems at all levels. It does not differentiate between top-level subsystems and lower-level subsystems. It is only concerned with connecting import and export objects, regardless of their owning subsystem's position in the hierarchy of subsystems in the application.

4.4 Modification of Event Flow

Section 3.5.4 developed how events flow through the subsystems during the execution of an application. In essence, events are passed to top-level subsystem's InEvent areas. When the Update function of the top-level subsystem is called, the top-level subsystem selects an event to process, and the routing scheme of the event is used to pass the event down the hierarchy of the independent subsystem to the consuming subsystem. Similarly,

when events are generated by low-level primitive objects, the event is passed back up the hierarchy, creating the routing scheme for the new events. The routing of events up and down the hierarchy of the independent subsystem is not very efficient in terms of execution time.

4.4.1 Solution. Since each subsystem has an InEvent area and an OutEvent area, the routing scheme is an unnecessary attribute of an event. The routing scheme can be replaced with the consuming subsystem. When an event is serviced by the application executive, it can be passed directly to the InEvent area of the subsystem that consumes it. When new events are generated at a low-level subsystem, they can be sent directly from the OutEvent area of that subsystem to the event manager. Depending on the number of levels of nested subsystems within an independent subsystem, this would minimize execution time, reduce complexity, and decrease memory usage.

4.4.2 Implementation. To pass events directly to/from the consuming/producing subsystems, the first step is to change the routing scheme attribute of an event from a sequence of subsystems to a single subsystem, representing the consuming subsystem. Figure 4.4 shows this modification. This figure uses the example subsystem of Figure 3.10 to illustrate the structure of an event before and after this modification. From the figure, it is easy to see that the current implementation uses the routing scheme information encapsulated in an event's data structure to pass the event down the hierarchy of the independent subsystem to the consuming subsystem. As composed applications become more complex and the number of nested subsystems in an independent subsystem increases, the number of subsystems in the routing scheme sequence increases, directly increasing execution time

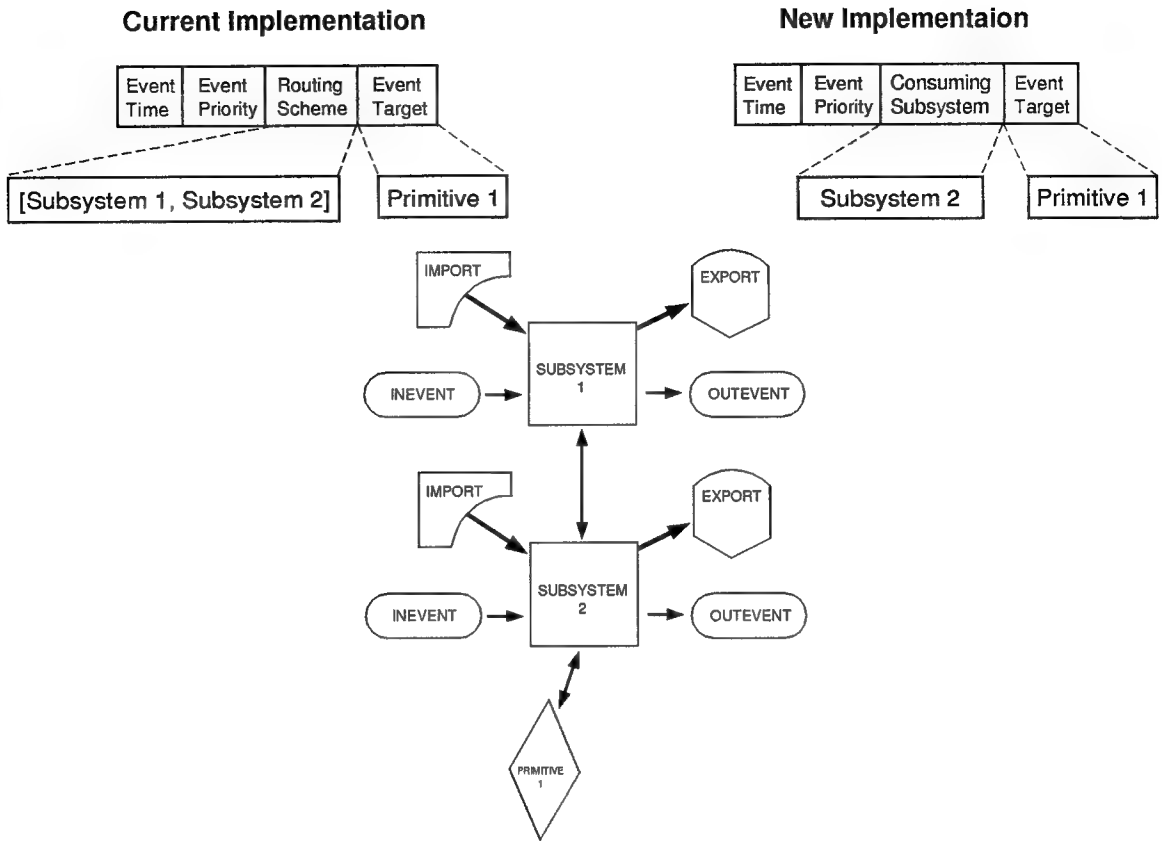

Figure 4.4 Modified Event Structure

and memory usage. The data structure of an event in the new implementation shows how the event is passed directly to the consuming subsystem. In this implementation, the execution time and memory required to store this data structure remain constant regardless of the complexity of the composed application, greatly simplifying event passing within Architect. Furthermore, subsystems in Architect are declared to have unique names; therefore, including the name of the consuming subsystem is all that is needed to use built-in Refine^{TM} object management functions to uniquely identify the appropriate InEvent and OutEvent area.

Next, the code that processes the routing scheme and routes events within an independent subsystem needs to be modified. The code responsible for this is the *event-driven-controller* function. This function is used for both event-driven and time driven modes.

Currently, the *event-driven-controller* function is passed a subsystem as an input parameter. The algorithm for this function is as follows:

1. get the next event from the InEvent area
2. if the size of the routing scheme sequence equals one (the event is consumed by this subsystem) then
   (a) determine the type of event (SetState, Update, etc.) and call the appropriate function of the target primitive
   (b) put newly generated events in the OutEvent area
3. else (the event is for a subordinate subsystem)
   (a) remove the first subsystem from the routing scheme
   (b) find the subsystem now at head of the routing scheme and place the event in its InEvent area
   (c) recursively call the *event-driven-controller* function
4. for all SetState events in the OutEvent area, prepend current subsystem to the routing scheme (creates the routing scheme as the events traverse back up the subsystem hierarchy after returning from each recursive call)

5. if the execution mode is time-drive then remove all application events from the Out-Event area (since time-driven primitives are not supposed to generate application events as presented in Section 3.6.2)

6. return the events in the OutEvent area to the executive

By passing events directly to/from the consuming/producing subsystems, the *event-driven-controller* function is simplified significantly. The function still receives a subsystem as its input parameter; however, the input parameter is now the consuming subsystem of the event. In addition, since time-driven primitives do not generate application events, the step in the above algorithm that removes application events from the OutEvent area in the time-driven mode is unnecessary. The new algorithm is as follows:

1. get the next event from the InEvent area

2. determine the type of event (SetState, Update, etc.) and call the appropriate function of the target primitive

3. put newly generated events in the OutEvent area

4. return the events in the OutEvent area to the executive

This new algorithm is significantly less complex and will execute faster than the current implementation. In addition, it is not dependent on the other modifications already presented in this chapter. It can be implemented, with equal success, either individually or in conjunction with the the other modifications.

4.5 NewData Event

In addition to the modifications already presented in this chapter, one minor change should be made. Section 3.5.1 stated that the distinction between application events and executive events lies in which subsystem does the actual processing of the event. With this in mind, the NewData event should be an application event since NewData events are processed by application subsystems, not executive subsystems. To implement this change, all that is needed is a simple change to the code declaring the OCU domain model. Instead of declaring the NewData event to be a subtype of executive event, it needs to be declared as a subtype of application event.

4.6 Summary

This chapter analyzed the current implementation of Architect's simulation environment and identified areas in which it could be modified to improve Architect's execution capabilities. In particular, three major modifications were identified. First, in order to provide a standardized method of exchanging data between subsystems and to restrict subsystems' visibility, connection objects need to exist between all export objects and the import objects they supply data for. Next, in order for Architect to conform to the OCU model, import and export areas need to be returned to all subsystems, not restricted to top-level subsystems. Finally, event routing needs to be modified so that events can be passed directly to/from the consuming/producing subsystem.

In addition to identifying areas requiring modification, this chapter provided a solution and a high-level discussion on how to implement each of the identified modifications. Once implemented, these modifications will reduce the complexity of Architect, and the

execution time of composed applications will be reduced. Building on the improvements already introduced, Chapter V presents areas in which the current simulation environment of Architect can be further extended.

V. *Extension of the Application Executive*

5.1 Introduction

Chapter IV analyzed the current implementation of Architect and identified areas in which the simulation environment could be improved. This chapter takes the analysis of Architect's simulation environment one step farther and looks at ways in which it can be extended. In particular, it examines mixed-mode execution and visually composable executives.

5.2 Mixed-Mode Execution

In order to support mixed-mode execution, Architect must be able to support multiple application executives within a single application. Before proceeding any further, it is important to present a brief justification as to why Architect should provide this capability. If Architect is going to be successfully used to simulate real, large-scale systems, it needs to be able to support the composition and execution of systems that are composed of subsystems from multiple domains with different execution modes. For example, consider the software system required for a modern military aircraft. The overall system would be composed of many subsystems to include a navigation subsystem and a radar warning subsystem. The navigation subsystem needs to respond to regular increments in the clock and calculate new position, heading, elevation, etc. Therefore, this subsystem would be time-driven. The radar warning subsystem needs to respond to asynchronous events raised by the environment and would be event-driven. In addition, a concurrent research effort by Harris (12) used object-oriented database management system (OODBMS) support to

compose applications from different domains; however, the applications created were restricted to a single mode of execution. It follows that the next extension should not only allow multiple domains but also multiple execution modes.

5.2.1 Alternatives. While analyzing this extension, two alternatives were considered. The first alternative was to allow every subsystem in the application to be executed by a separate application executive, and the second approach was to allow only independent subsystems to be executed by separate executives. In this case, the execution mode of the top-level subsystem would determine the execution mode of all subordinate subsystems.

In order to select between these two alternatives, it was necessary to identify some evaluation criteria upon which to base this decision. The decision was based on the following:

- size (memory usage)
- complexity
- flexibility
- ease of implementation

The first criterion considered was size. As composed applications become larger and more complex, it is possible that the applications could be limited by the amount of available memory. Of the two alternatives considered, the second (independent subsystems with different execution modes) would reduce the potential number of application executives in the composed applications and therefore, use less memory.

The second criterion considered was complexity. It is desirable to minimize the complexity of the system. As the number of application executives in the application

increases, the complexity of the application increases. With this in mind, the second alternative is once again the favored choice.

The third criterion considered was flexibility. Flexibility refers to the ability to accurately model any real-world application. The advantage here goes to the first alternative, since there may be applications that would require each subsystem in an application to operate in a different execution mode; however, you could circumvent this using the second alternative by making every subsystem in the application a top-level subsystem and therefore, an independent subsystem.

The final criterion considered was ease of implementation. Ease of implementation follows directly from complexity. The less complex the system, the easier it is to implement. Since the second alternative is less complex, it is also easier to implement.

5.2.2 Independent Subsystems with Executives. After evaluating the criteria presented in the previous section, it was decided that mixed-mode execution would best be implemented using the second alternative, namely, allowing a different execution mode for each independent subsystem. The initial step required for this extension of Architect was to create an object model of the new system. Figure 5.1 shows the object model that was developed to support mixed-mode execution, allowing multiple application executives within an application. This object model is very similar to the event-driven model presented in Figure 3.1. The main differences are the application may now have multiple executives and the addition of a *time manager* subsystem.

In the new object model, the application is composed of one or more composition units. Each composition unit (abstract class) can connect to zero or more composition

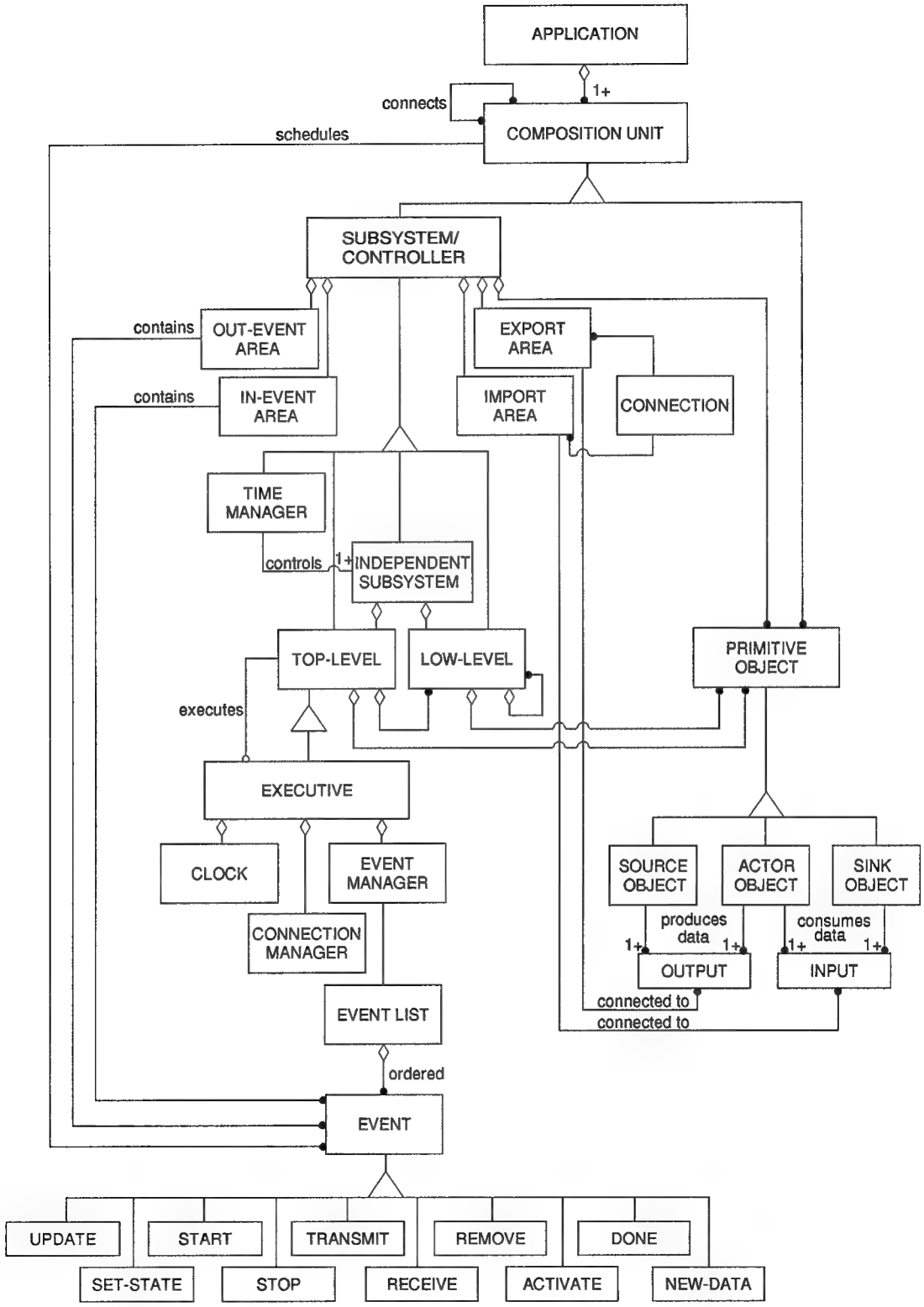

Figure 5.1 Object Model of an Application Supporting Multiple Executives

units and is either a primitive object or a subsystem (concrete classes). All subsystems are composed of an import area, export area, in-event area, out-event area, and zero or more primitive objects. An independent subsystem is a subsystem that is composed of top-level and low-level subsystems. Top-level subsystems and low-level subsystems are in turn composed of zero or more low-level subsystems and zero or more primitives. The executive is a top-level subsystem, and there is zero or one executive instantiated to execute each top-level application subsystem (there will not be an executive for independent subsystems that execute in non-event-driven-sequential mode; the subsystems associated with this mode will have an update algorithm as presented in Chapter II). Finally, the time manager is a subsystem that controls the order in which the independent subsystems in the application execute.

Figure 5.2 shows a block diagram of a composed application supporting mixed-mode execution. This is a high-level diagram of an application; the independent subsystems and executives would also have primitive objects and/or other low-level subsystems subordinate to them. This diagram is provided mainly to illustrate a big-picture view of how the time manager fits into the hierarchy of a composed application.

5.2.3 Concept of Operation. Once the object model was specified, it was necessary to define how an application would execute with this new capability. There were two major areas that needed to be addressed: the advancement/management of time and data consistency between independent subsystems. These are discussed in the following sections.

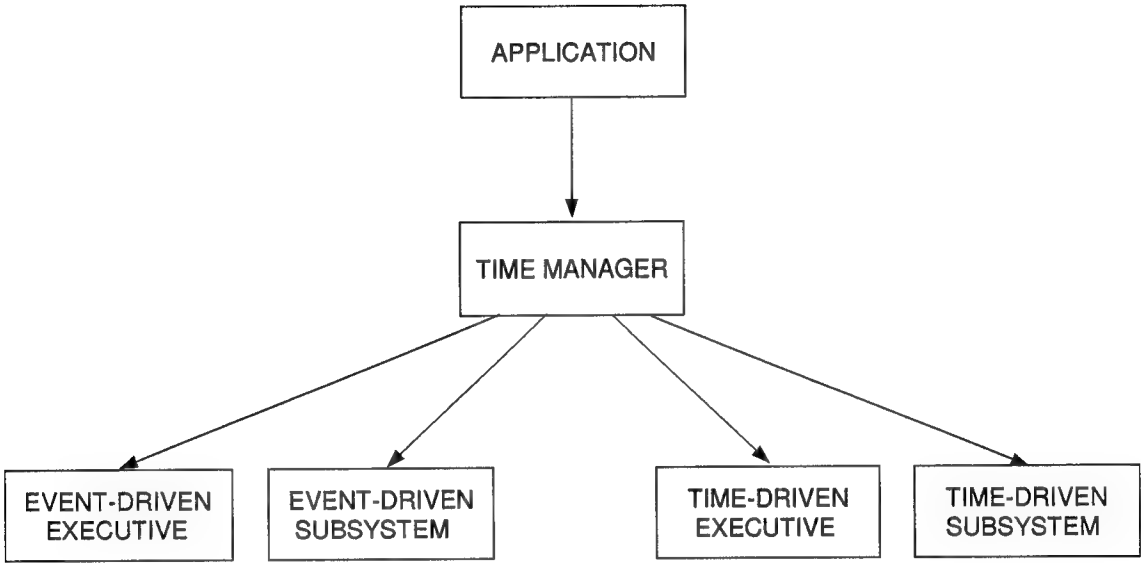

Figure 5.2 Block Diagram of an Application Supporting Mixed-Mode Execution

5.2.3.1 Time Management. The time manager subsystem was incorporated into the object model to handle this function; however, it was necessary to define exactly how the time manager would provide this service. Since the overall execution of the application is to remain sequential, the time manager needed to know the simulation time of the next event for each executive in the application and pass control to the one with the next simulation time. In order to do this, the time manager needs to maintain a list containing one entry for each executive in the application. Each entry, in turn, needs to include the executive name and the time of the next event that the executive is to service. When the Update function of the time manager is called, it searches the list for the smallest time value, t, and then calls the Update function of the appropriate executive.

The executive services all events in its event list for the given simulation time (all events originally in the event list scheduled for time t and any events generated for time t as a result of servicing other events). When the called executive is finished servicing all time t events, it returns the time of the next event to be serviced to the time manager and control is then given back to the time manager. The time manager once again selects the executive with the smallest next event time, and the cycle continues.

5.2.3.2 Data Consistency. The next major area of concern was how to maintain data consistency as data is passed between independent subsystems. The steps necessary to maintain data consistency vary, depending on the execution mode of the producing (source) subsystem and the consuming (target) subsystem. There were two cases that needed to be considered. The first case is when the target subsystem's execution mode is event-driven, and the second is when the target subsystem's execution mode is time-driven.

The first case is the simpler of two and will be examined first. In this scenario, the target subsystem's execution mode is event-driven, and the source subsystem is either time-driven or is under the control of a *different* event-driven executive. In either case, the servicing of a Transmit event in the source subsystem's executive will use a connection object to pass the data from the source export object to the sink import object. The executive will then generate an Update event for insertion into the target subsystem's associated executive using the source executive's current time. Because of the way in which the time manager controls the execution of the independent subsystems (see Section 5.2.3.1), the time of the new Update event can not be earlier than the next event in the target exec-

utive's event list. In addition, the insertion of the Update event must be done differently than it was in the previous implementation. Previously, when an event was inserted into the event list, the executive was passed an offset, and event manager calculated the absolute simulation time prior to insertion into the event list. This will still work if the source export and sink import belong to the same independent subsystem; however, when they do not, the absolute simulation time is passed to the target subsystem's executive, and the executive tags the new event with this time. Finally, when the controlling executive finishes servicing all events at the current simulation time, it will return the time of the next event and return control to the time manager. If the controlling executive is event-driven, it will return the time of the next event in the event list. If the controlling executive is time-driven, it will return a time equal to the current time plus Δt, where Δt equals one clock tick.

In the second case, the target subsystem's execution mode is time-driven, and the source subsystem is either event-driven or is under the control of a *different* time-driven executive. In this scenario, there needs to be two import objects associated with each of the target primitive's inputs: one labelled *old* and the other labelled *new*. In addition, the data structure of these import objects needs to include a new field representing *time*. When a transmit event is serviced in the source subsystem's executive, the new data will be written to the target primitive's *new* import object and will be stamped with the current time of the source executive. When the target executive (time-driven) receives control from the time manager and one of its primitives updates, it must compare its current time to the time of the associated *new* import object. If the current time is less than the time of the *new* data, then it will use the *old* data (the *new* data should not yet be visible to

this primitive). If the current time is equal to or greater than the *new* data, then use the *new* data and copy the *new* import object to the *old* import object.

5.3 Composable Executives

In the current implementation, the event-driven and time-driven application executives are instantiated textually and saved to a file. Each time a new application is composed requiring one of these execution modes, the associated file representing the appropriate execution mode is parsed into the application, and then the application specialist composes the new application. There was no way to use AVSI to compose application executives. Previous research (32) recommended the visualization of the application executive as an area of further research. In addition, as Architect evolves, the need for composable executives will surely become a necessity. Even though the technology base for the application executive currently contains only five primitives, any real system developed using this technology would require more specialized primitives. For example, in the time-driven domain, different applications would need different clock resolutions. There could be numerous time-driven clock primitives, and the application composer would select the appropriate one during the composition of the executive subsystem.

In a previous research effort, Warner (30) developed a methodology for populating Architect's knowledge base. This methodology provided a systematic way in which to incorporate composable executives into Architect. In addition, since the event-driven and time-driven executives had already been instantiated textually, many of the required steps presented in Warner's methodology had already been accomplished.

5.3.1 Visualization of the Executive Primitives. At a high level, there are two steps required to instantiate a domain (populate and visualize the primitives) in Architect. The first step is to perform a domain analysis on the domain, and the second step is to implement the domain model developed in Step 1. As presented in Section 3.3, Welgan had already performed the domain analysis. The implementation of the domain model in Architect, according to Warner, is further broken down into the following six steps:

1. declare the domain model classes
2. define the attributes for the domain classes
3. declare domain-specific types and functions
4. define the domain specific language (DSL)
5. define the visual specification objects (VSO) that are parsed in by the visual specification language (VSL)
6. declare the icon objects and draw the icons

In the process of instantiating the executive domain primitives textually, the first four of these steps had already been accomplished, therefore, only the last two were required to visualize the executive primitives.

The VSOs were defined in a file that, when parsed by the VSL grammar, instantiates the VSOs AVSI requires to manipulate domain primitives. In other words, the VSL is used to load the information necessary to visualize the domain. An example portion (representing the time-driven clock primitive) of the executive domain VSL specification is shown in Figure 5.3. The *Icon* section is the same for every primitive, except for the *td-clock-bitmap-l* and the *td-clock-bitmap-s* which are the specific names of the icon objects

for this primitive. The *Edit* section lists the attributes of the primitive that can be changed by the application composer. If an attribute of a primitive is not listed in this section, then it will not be listed when the application composer selects the menu choice that allows attribute values of a primitive to be changed.

```
attributes for TD-CLOCK-Obj are
Icon :
    label = class-and-name;
    clip-icon-label? = false;
     border-thickness = 0;
     bitmap4icon-l = td-clock-bitmap-l;
     bitmap4icon-s = td-clock-bitmap-s
Edit :
    name : symbol;
    time : integer
end;
```

Figure 5.3 Time-Driven Clock Portion of the VSL Specification

All that remained to visualize the executive domain's primitives was to declare the icon objects and draw the icons. Architect creates the icons it uses from bitmaps, which are pixel-by-pixel representations of graphics. The only information needed for these definitions is the icon object names and the file names for the icon bitmaps. Figure 5.4 shows the associated icon object definitions created for the time-driven clock primitive.

```
%% variables to hold the icon's bitmaps (-l is 64x64, -s is 32x32)

var td-clock-bitmap-l : any-type=(cw::read-bitmap("EXECUTIVE-TECH-BASE/td-clock.icon-l"))
var td-clock-bitmap-s : any-type=(cw::read-bitmap("EXECUTIVE-TECH-BASE/td-clock.icon-s"))
```

Figure 5.4 Time-Driven Clock Icon Object Definitions

As stated above, icons in Architect are made from bitmaps, and every icon has a large (-l) and a small (-s) form. The large form is a 64x64 bitmap and the small form is a 32x32 bitmap, and they were created using the Open Windows^{TM} IconEdit tool. Figure 5.5

shows the technology base window for the executive domain and the associated icons that were developed.

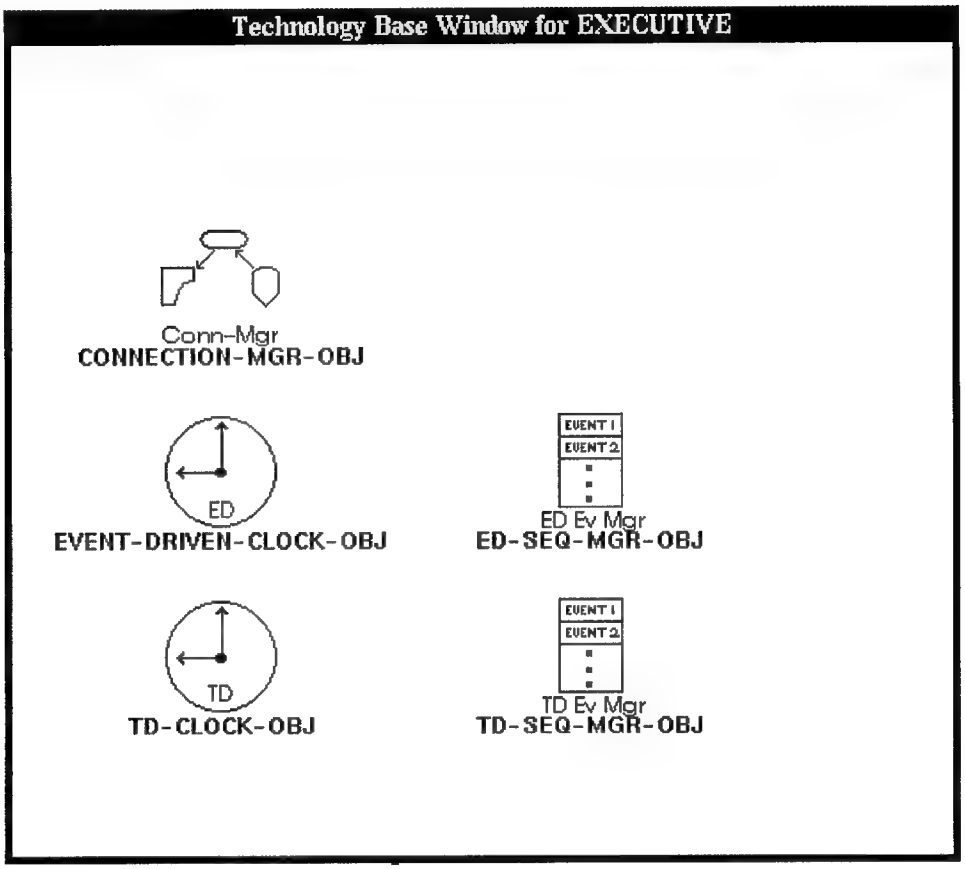

Figure 5.5 Visualization of the Executive Domain Primitives

5.3.2 Concept of Operation. Once the executive primitives were visualized, it was necessary to define a concept of operations for applications with composable executives. The subsequent narrative defines the steps to be followed during the composition of an application with composable executives.

The application specialist first selects the domain in which to compose the application. Once the domain is selected, Architect provides a menu, prompting the composer to

select a saved executive or create a new one. If he/she chooses to use a saved executive, a menu listing the saved executives for the execution mode of the selected domain will be displayed by AVSI, and the composer selects an executive from this list. Once a selection is made, the associated executive will be parsed in and displayed in the System Composition Window. Next, the Technology Base Window for the selected application domain will be displayed, and the composition process will continue as currently implemented.

If the application specialist chooses to create a new executive, the Technology Base Window will first display the executive primitives, and the composer will assemble the executive subsystem in the same manner that application subsystems are created. When finished with the executive, the composer needs to inform AVSI to display the application domain primitives in the Technology Base Window. This is done by clicking on the Technology Base Window and selecting "Display Application Domain Primitives" from the resulting menu. Once the associated application domain primitives are displayed, the composition process will continue as currently implemented. In addition, the capability to modify an existing executive is needed; therefore, the menu displayed as a result of clicking on the Technology Base Window will also provide an option to "Display Executive Primitives". This gives the application specialist the flexibility to alternate between the executive subsystem and the application subsystems during the composition of an application.

Figure 5.6 shows the System Composition Window visualizing a simple application from the event-driven circuits domain.

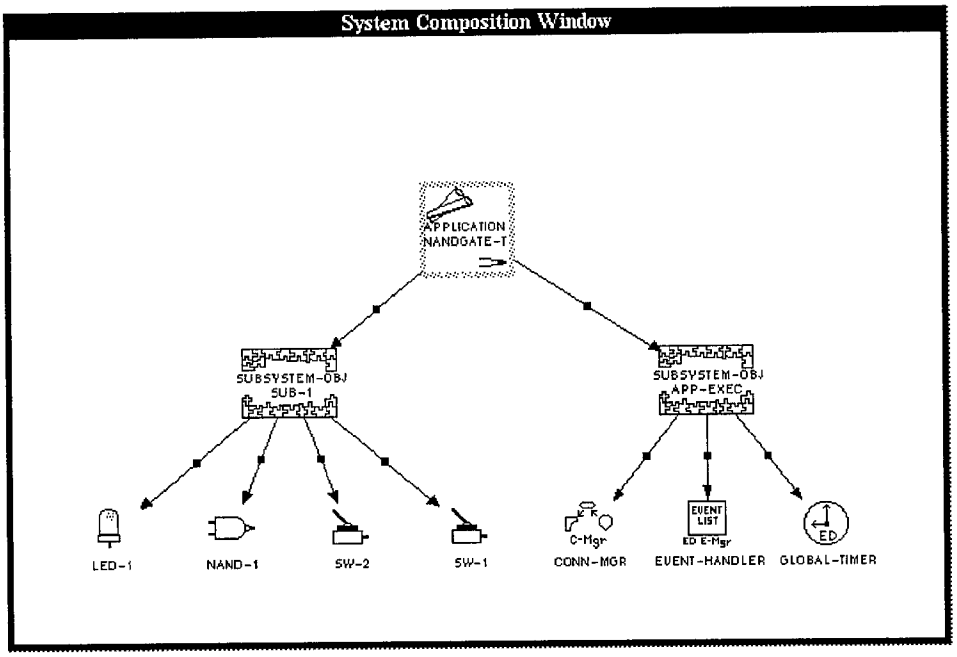

Figure 5.6 Visualization of the Executive in a Simple Application

5.3.3 Implementation. To complete the integration of the application executive with the composition kernel, all the remaining changes necessary to implement the concept of operation as defined in Section 5.3.2 consist of modifications to AVSI code. One additional implementation issue concerns the saving of an application executive subsystem. Currently, an application is saved by transforming the Refine^{TM} abstract syntax tree (AST), rooted at the application object, into a file written in a domain specific language (DSL). In order to save an application executive subsystem, the AST rooted at the executive subsystem needs to be saved to a file, not the AST for the entire application.

5.4 Summary

This chapter developed two extensions to the current implementation of the Architect system. First, it discussed the incorporation of mixed-mode execution. An object model

was developed showing how the new implementation differs from the current implementation, in particular, the addition of a time manager subsystem and the existence of multiple executives within an application. Following the specification of the object model, a concept of operation was developed that addressed two major areas of interest: time management and data consistency. Next, the incorporation of visually composable executive subsystems was presented. Warner's methodology for populating Architect's knowledge base (30) was used to aid in visualizing the executive primitives. Finally, Chapter VI provides the conclusions of this research effort and recommended areas for further research.

VI. Conclusion and Recommendations

6.1 Overview

This chapter provides a summary of the accomplishments of this research effort. It begins with a review of the original goals and then compares them to the final results. It also presents several recommendations for further research and discusses conclusions that can be drawn from this work.

6.2 Research Goals

The original goals of this thesis, as defined in Chapter I, were as follows:

1. To develop an indepth understanding of the current implementation of the application executive.
2. To consolidate the information detailing the implementation of the application executive into a single, comprehensive document.
3. To perform a critical evaluation of the application executive, analyzing the overall operational concept and the domain model for the event-driven and time-driven sequential operating modes.
4. To analyze Architect to determine changes required for independent subsystems to execute in different modes.
5. To complete the integration of the application executive with the composition kernel and integrate the application executive with AVSI.

6.3 Results

This research was successful in achieving its objectives. Chapter III provides a thorough description of the current implementation of Architect's simulation environment. Information disseminated throughout several thesis reports and Architect's source code was combined into a single, comprehensible document, enhancing the overall understandability of the system. Next, a critical evaluation identified areas in which the current implementation could be modified to improve Architect's execution capabilities. In particular, three major modifications were identified:

1. Provide a standardized method in which to exchange data between subsystems and restrict subsystems' visibility by incorporating connection objects between all import and export objects.
2. Re-establish conformity with the OCU model by enabling all subsystems to have import and export areas.
3. Modify event flow so that events can be passed directly to/from the consuming/producing subsystem.

In addition to identifying areas requiring modification, implementation details were developed for each. Once implemented, the modifications will reduce the complexity and execution time of applications composed in Architect. Next, an object model and concept of operations were developed to illustrate how Architect could be modified to support mixed-mode execution. The concept of operations addressed the important issues of time management and data consistency. Finally, this research visualized the application execu-

tive primitives and presented a concept of operations for completing the integration of the application executive with AVSI.

6.4 Recommendations for Future Research

- *Implement Critical Evaluation Solutions.* As mentioned in the previous section, the critical evaluation identified three modifications that would improve the execution capabilities of Architect. This thesis effort was primarily an analysis of the current implementation, therefore, further research should carry the analysis one step further and implement the modifications of connection objects, import/export areas, and event flow. The actual implementation should be rather straight forward since this thesis provided implementation details as well as a solution to each of these.

- *Implement Mixed-Mode Execution.* This research developed the groundwork on which to implement mixed-mode execution. The object model and concept of operations illustrated the feasibility of mixed-mode execution and offer a foundation on which to extend Architect to include this capability.

- *Complete Integration of the Executive with AVSI.* The integration of the application executive with AVSI is nearly complete. The technology base for application executive primitives has been visualized, and all that remains to complete the integration of the executive with the composition kernel is to modify AVSI code to implement the concept of operation as presented in Section 5.3.2.

- *Concurrent Simulation.* The event-driven sequential and time-driven sequential simulation capabilities within Architect allow for only a single thread of control within

an application. This needs to be extended to include event-driven concurrent and time-driven concurrent modes of execution. A decision would have to be made as to what level of concurrency would be allowed. The level of concurrency must address whether or not only top-level independent subsystems could execute concurrently or whether subsystems subordinate to the top-level subsystems would be allowed to execute concurrently.

6.5 Conclusions

This thesis effort significantly improved the documentation of the simulation environment of Architect. In addition to providing a more understandable description, the documentation removed any inconsistencies between the thesis reports from which it was derived. The critical analysis identified problem areas with the current implementation and presented solutions to these problems. Once implemented, the solutions will enhance Architect's execution capabilities by reducing complexity and application execution time. The development of an object model and concept of operation for mixed-mode execution provide the groundwork upon which to extend Architect's simulation capabilities. Finally, the integration of the application executive with AVSI is nearly complete. The executive primitives were visualized, and a concept of operation was provided to define how applications with composable executives would be implemented. All that remains is to modify AVSI code to implement the concept of operation.

Bibliography

1. Anderson, Cynthia. *Creating and Manipulating Formalized Software Architectures to Support a Domain-Oriented Application Composition System.* MS thesis, AFIT/GCS/ENG/92D-01, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.

2. Arango, Guillermo. "Domain Analysis, From Art Form to Engineering Discipline." *Domain Analysis and Software Systems Modeling* edited by Guillermo Arango and Rubén Prieto-Díaz, 81–88, IEEE Computer Society Press, 1991.

3. ASC/RWWW. *System Concept Document for the Joint Modeling and Simulation System (J-MASS) Program.* CROSSBOW-S Architecture Technical Working Group, November 1991.

4. Banks, Jerry and II John S. Carson. *Discrete Event Simulation.* Englewood Cliffs, New Jersey: Pentice Hall, Inc, 1984.

5. Cecil, Danny A. and Joseph A. Fullenkamp. *Using Object-Oriented Database Management System Technologies to Archive Formally Specified Software Artifacts for Re-Use in a Domain-Oriented Application Composition Environment.* MS thesis, AFIT/GCS/ENG/93D-03, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

6. Cossentine, Jay. *Developing a Sophisticated User Interface to Support the Architect Domain-Oriented Application Composition and Generation Systems.* MS thesis, AFIT/GCS/ENG/93D-04, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

7. D'Ippolito, Richard S. "Using Models in Software Engineering." *Proceedings: TRI-Ada '89.* 256–265. New York, NY: Association of Computing Machinery, Inc., 1989.

8. D'Ippolito, Richard S. and Charles P. Plinta. "Software Development Using Models." *Proceedings: Fifth International Workshop on Software Specication and Design.* 140–142. New York, NY: Association of Computing Machinery, Inc., 1989.

9. Gool, Warren E. *Alternative Architectures for Domain-Oriented Application Composition and Generation Systems.* MS thesis, AFIT/GCS/ENG/93D-11, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

10. Graybeal, Wayne J. and Udo W. Pooch. *Simulation: Principles and Methods.* Cambridge, MA: Winthrop Publishers, Inc., 1980.

11. Guinto, Richard A. *Visualization and Manipulation of the Architect Visual System Interface.* MS thesis, AFIT/GCS/ENG/93D-04, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1994.

12. Harris, Alfred W., Jr. *Using Object-Oriented Database Technology to Develop a Multiple Domain Capability for Domain-Oriented Application Composition Systems.* MS thesis, AFIT/GCS/ENG/94D-07, School of Engineering, Air Force Insitute of Technology (AU), Wright-Patterson AFB, OH, December 1994.

13. IEEE. *IEEE Standard VHDL Language Reference Manual.* New York, NY, 1988. IEEE Std 1076-1987.

14. Iscoe, Neil. "Domain-Specific Reuse: An Object-Oriented and Knowledge-Based Approach." *Tutorial on Software Reuse: Emerging Technology* edited by Will Tracz, IEEE Computer Society Press, 1988.

15. Khoshnevis, Behrokh. *Discrete Systems Simulation.* San Francisco, California: McGraw Hill, Inc, 1994.

16. Lee, Kenneth J. and others. *Model-Based Software Development (Draft).* Technical Report CMU/SEI-92-SR-00, Software Engineering Institute, December 1991.

17. Lipsett, Roger and others. *VHDL: Hardware Description and Design.* Norwell, MA: Kluwer Academic Publishers, 1989.

18. McCain, Ron. "Reusable Software Component Construction: A Product-Oriented Paradigm." *AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference.* 125–135. AIAA, October 1985.

19. Neighbors, James M. "Draco: A Method for Engineering Reusable Software Systems." *Domain Analysis and Software Systems Modeling* edited by Guillermo Arango and Rubén Prieto-Díaz, 34–52, IEEE Computer Society Press, 1991.

20. Perry, Douglas L. *VHDL.* San Francisco, CA: McGraw-Hill, Inc., 1991.

21. Prieto-Díaz, Ruben. "Domain Analysis for Reusability." *Proceedings of the COMPSAC '87.* 23–29. 1987.

22. Prieto-Díaz, Ruben. "Domain Analysis: An Introduction," *ACM SIGSOFT Software Engineering Notes*, *15*(2):47–54 (April 1990).

23. Randour, Mary Anne. *Creating and Manipulating a Domain-Specific Formal Object Base to Support a Domain-Oriented Application Composition System.* MS thesis, AFIT/GCS/ENG/92D-13, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.

24. Reasoning Systems, Inc. INTERVISTA^{TM} *User's Guide.* Palo Alto, CA, 1991. For INTERVISTA^{TM} Version 1.0.

25. Reasoning Systems, Inc. *DIALECT User's Guide.* Palo Alto, CA, July 1990.

26. Reasoning Systems, Inc. *REFINE User's Guide.* Palo Alto, CA, May 1990.

27. Rumbaugh, James and others. *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey: Pentice Hall, Inc, 1991.

28. Tracz, Will and others. "A Domain-Specific Software Architecture Engineering Process." From Collected Papers of the Domain-Specific Software Architectures (DSSA) Avionics Domain Application Environment (ADAGE), May 1993.

29. Waggoner, Robert W. *Domain Modeling of Time-Dependent Systems.* MS thesis, AFIT/GCS/ENG/93D-23, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

30. Warner, Russel M. *A Method for Populating the Knowledge Base of AFIT's Domain Oriented Application Composition System.* MS thesis, AFIT/GCS/ENG/93D-24, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

31. Weide, Timothy. *Development of a Visual System Interface to Support a Domain-oriented Application Composition System.* MS thesis, AFIT/GCS/ENG/93M-05, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, March 1993.

32. Welgan, Robert L. *Domain Analysis and Modeling of a Model-Based Software Executive.* MS thesis, AFIT/GCS/ENG/93D-25, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

33. Whitted, Gary A. *Software Development Plan for the J-MASS Modeling Components - Vol II.* Architectural Technical Working Group, ASD/RWWW, April 1992.

Vita

Captain Keith E. Lewis was born on November 28, 1961 in Missouri Valley, Iowa and graduated from Missouri Valley High School in 1980 as valedictorian of his graduating class. He attended the University of Iowa in Iowa City, Iowa for one year and then enlisted in the Air Force in August, 1981. Following basic training, he completed technical training for F-111 avionics maintenance at Lowry AFB, Colorado, in April, 1982 and was assigned to the 366th Component Repair Squadron, Mountain Home AFB, Idaho. He was accepted into the Airman Education and Commissioning Program and returned to the University of Iowa in January, 1985. Upon graduating with a Bachelor of Science degree in Electrical Engineering in December, 1987, he attended Officer Training School (OTS) at Lackland AFB, Texas. He graduated as a distinguished graduate from OTS and received his commission on June 23, 1988. As a new lieutenant, he was assigned to the 544th Intelligence Wing, Offutt AFB, Nebraska, as an electronic intelligence engineer. In May, 1993, Captain Lewis entered the Air Force Institute of Technology at Wright-Patterson AFB, Ohio, to pursue a Master of Science degree in Electrical Engineering. Upon graduation, Captain Lewis will be assigned to the Air Force Technical Applications Center, Patrick AFB, Florida, as a computer engineer in the nuclear treaty monitoring directorate.

Permanent address: 1881 Crane Creek Blvd
Melbourne, FL 32940